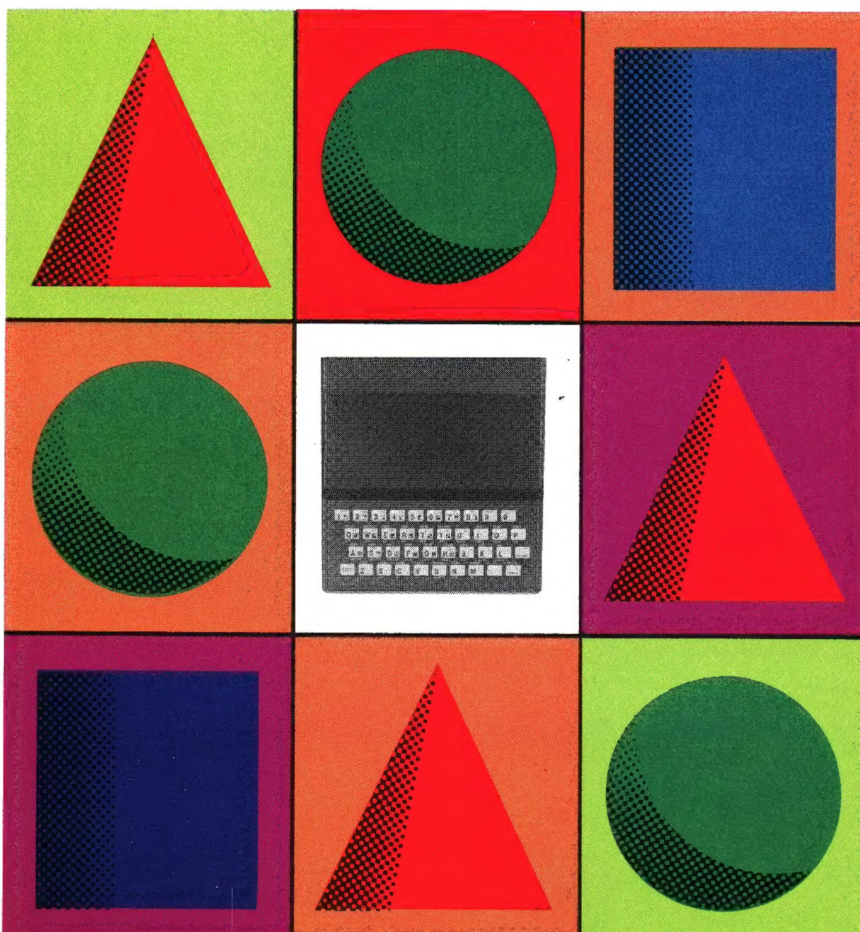
22012

TIMEX SINCLAIR 1000™/ZX81 USER'S HANDBOOK

By Trevor J. Terrell & Robert J. Simpson



Timex Sinclair 1000/ZX81 User's Handbook

by
Trevor J. Terrell and
Robert J. Simpson

Formerly titled
ZX81 User's Handbook

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

FIRST EDITION
FIRST PRINTING — 1983

First published in the United Kingdom by Newnes Technical Books, an imprint of the Butterworth Group. Americanized version published by Howard W. Sams & Co., Inc.
Indianapolis, IN 46268

International Standard Book Number:
(United Kingdom Edition) 0-408-01223-4
(North American Edition) 0-672-22012-1
Library of Congress Catalog Card Number: 82-61061

North American Edition
Edited by: *Welborn Associates*
Illustrated by: *David K. Cripe*

Printed in the United States of America.

Preface

Unless you are a computer specialist it is quite likely that you will want answers to a number of questions concerning your computer. In this book we provide answers to many and varied questions asked by students, school children, hobbyists, and teachers.

The range of questions answered covers the operation of the computer and its peripherals, troublesome aspects of BASIC® programming, principles of machine-code programming, hardware details, and principles of interfacing user hardware via the edge connector. When appropriate, answers are supported with suitable BASIC or machine-code programs, thereby enabling you to gain hands-on experience and to investigate and verify points explained in the text. Furthermore we hope that you will find this book a useful source of reference when programming and using your Timex Sinclair 1000/ZX81 computer.

We wish to thank Sue Wasson for her hard work and competence in typing the manuscript. We are grateful to Mostek Corporation for their kind permission to use the Z80A Instruction Set Summary from their Data Book.

We sincerely thank our wives Meryl (S.) and Jennifer (T.) for their unfailing support and encouragement.

R.J.S.
T.J.T.

Dedicated to Balaclava Joe

In August, 1982, the Sinclair ZX81 computer became the TIMEX SINCLAIR 1000™*. It is the same computer, but with a 2K RAM as opposed to a 1K RAM in the ZX81. Throughout this book any references made to the ZX81 also apply to the TIMEX SINCLAIR 1000.

*Trade name of Timex Computer Corp.

Contents

CHAPTER 1

Please Sir . . .

Now I have my Timex Sinclair 1000/ZX81, how do I start?

All you need do is connect the 9V dc power supply (the jack plug) to the ZX81, and the uhf tv cable from the output of the ZX81 to the uhf antenna input of your black and white or color tv set, as shown in Fig. 1-1.

After switching on both the dc supply and your tv set, select a tv channel and tune it until the prompt or cursor (a white upper case K on a black square) is clearly displayed in the bottom lefthand corner of the screen. You will probably find it desirable to turn your tv volume control to its minimum setting when working with your computer.

As a simple test to establish that your computer is working, press the following keys in the order given. First press **PRINT**, then 7, then + (this is obtained by holding down the **SHIFT** key and pressing K), then 9, and finally **ENTER**. The computer then calculates the sum 7 + 9 and displays the result, 16, in the top lefthand corner of the screen. The displayed report code message, in the bottom lefthand corner of the screen, will be **0/0**, indicating successful completion of the instruction (Table 1-1).

Now that you have got started try a few more sums to help you become familiar with the keyboard and the display.

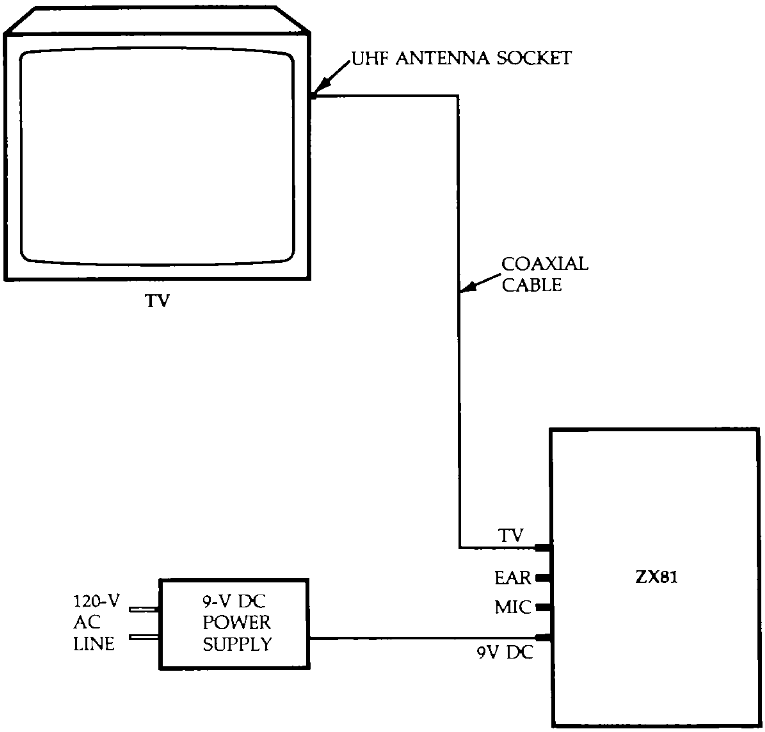


Fig. 1-1. Standard Timex Sinclair 1000/ZX81 system.

Report codes are displayed in the bottom lefthand corner of the screen in the form: Report code/Program line number

How do I use a cassette tape recorder?

The Timex Sinclair 1000/ZX81 stores programs and information in its volatile memory, which means that when you turn off the power supply the program and data will be lost. However, when you wish to keep a program for use at some future time, you can copy (save) it from the memory of your microcomputer onto cassette tape, and when you wish to run the program again you can copy (load) it from the cassette tape into the memory of the Timex Sinclair 1000/ZX81.

To transfer a program from your computer to a cassette tape you should first connect the pair of leads provided (they have 3.5 mm jack plugs at either end) from the EAR and

Table 1-1. Report codes

Report Code	*Meaning*
0	Completed successfully, or program has jumped to line number outside existing range
1	Control variable nonexistent, but ordinary variable has same name
2	Variable used is undefined
3	Subscript out of range
4	Insufficient memory
5	Screen is full
6	Arithmetic overflow
7	**GOSUB** omitted
8	**INPUT** not allowed
9	**STOP** statement executed
A	Function has invalid argument
B	Integer out of range
C	**VAL** expression not valid
D	**BREAK** executed
E	Not used
F	Program name is the empty string

MIC sockets on the Timex Sinclair 1000/ZX81 to the corresponding inputs on your cassette tape recorder. Then, with the tape in the position where you wish to start recording the program, type on the computer keyboard

SAVE "Name of Program"

but *do not* press the **ENTER** key. Note that we have assumed that you have a name for your program; if this is not the case then just type

SAVE " "

on the keyboard. Next start the cassette recorder recording and press the **ENTER** key. Note that the tv screen now displays irregular patterns but after a while this ceases and the report **0/0** appears on the screen, indicating that the program has been recorded. You can now stop the tape recorder having SAVEd your program.

When a program is saved, the latest data inputted or assigned to numeric variables is also recorded, consequently these are the initial data values assigned to the numeric variables immediately after the program is loaded.

To transfer a program from a cassette tape into your Timex Sinclair 1000/ZX81 you must position the tape so that

it is at the beginning of the program, ensure that the EAR socket on your computer is connected to the earphone socket on the cassette recorder, and adjust the volume control on the cassette recorder to about half to three-quarters of the maximum setting. Next, type on the computer keyboard.

LOAD "Name of Program"

but do not press the **ENTER** key. Alternatively, if your program does not have a name, type

LOAD " "

Next start the cassette recorder playing and press the **ENTER** key. Once the computer has loaded the program it responds with **0/0** and you can switch off your cassette recorder.

If you are anxious to try this facility of saving a program select any line from one of the programs in Chapter 8 and follow the procedure above after entering the program line through the keyboard. When you have successfully saved this line of program on cassette tape, momentarily remove the 9V dc supply to the Timex Sinclair 1000/ZX81 to destroy the line of program in the computer's volatile memory, and then the saved program line can be recorded back into the memory using the load procedure described above.

In some cases it is useful to have the **SAVE** "Name of Program" as a line of program. This will enable the program to save itself, and when this is accomplished the program will execute the next line—this is a SAVE and RUN process. A program that has been saved in this way may be loaded as described above, but you should note that after loading is complete, the program automatically starts executing from the line immediately following the **SAVE** statement—this is a LOAD and RUN process.

To verify the SAVE and RUN/LOAD and RUN operations insert two additional lines in Program No. 6 in Chapter 8, namely

145 **SAVE** "DIE"
155 **GOTO** 15

To SAVE and RUN the modified program, with the tape in position, enter **RUN** 145, start the cassette recorder record-

ing and press the **ENTER** key. To LOAD and RUN this program position the tape before the beginning of the program, enter **LOAD** "DIE", start the cassette recorder playing, and then press the **ENTER** key.

What do I do if the program does not load first time?

The volume setting on the cassette recorder is rather critical so, if your program does not load at the first attempt, change the volume setting and try again. You may have to try a few times before you achieve the correct signal level for loading your programs. If you fail to load your program after several attempts, try a different type of cassette recorder.

Why has the Timex Sinclair 1000/ZX81 an exposed edge connector?

The exposed edge connector on your Timex Sinclair 1000/ZX81 contains 44 of the circuit connections of the microcomputer. It contains all the pin connections of the Z80A microprocessor and special control lines (the actual hardware details are described in Chapter 9). These connections are made available so that extra peripheral devices can be added to your microcomputer.

The edge connector behaves as an expansion port so that the *Random Access Memory* (RAM—see Chapter 9) of your microcomputer can be increased in size from the standard value of 1K to an extended size of 16K. The Sinclair 16 Kbyte RAM pack has been designed for this task and it plugs directly onto the Timex Sinclair 1000/ZX81 or the accessory printer exposed edge connector. You may think that you will never need 16K of memory but as the sophistication of your programs increases you will probably find that more memory is essential.

Another system element that has been designed to plug directly on to the exposed edge connector is an accessory printer. The printer enables you to have a permanent record of anything that the computer can display on the tv screen, and it permits long programs and lists of results to be printed. The output from the printer consists of 32 columns,

the same as the output on the tv screen, with as many rows as you wish.

Since the edge connector contains all the pin connections of the Z80A microprocessor, it is possible to design interface and control circuitry which can be used to make your computer control external systems. This can be achieved using some of the commercially available interface devices or you may eventually wish to design and build your own.

In Chapter 9 we shall consider some aspects of the input/output (I/O) capabilities of the Timex Sinclair 1000/ZX81, and discuss the edge connector and the signals available, in much more detail.

CHAPTER 2

Key It In

How do I operate the keyboard?

You will notice that up to five characters, symbols or words are assigned to each of the 40 keys (Fig. 2-1). The computer displays a cursor to prompt you to key a valid entry.

After switching on your computer the K cursor is displayed, and the system then requires you to enter either a program line number (any positive integer in the range 1 to 9999) or a keyword (the words written above the individual keys). When entering programs the K cursor occurs automatically at places in the program line where the microcomputer is expecting a valid keyword.

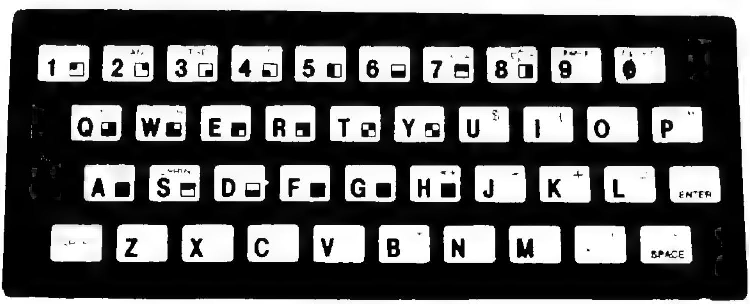


Fig. 2-1. Timex Sinclair 1000/ZX81 keyboard.

After entering a keyword the computer displays the L cursor to indicate that it is waiting for you to input an alphanumeric symbol (number or letter), **FULLSTOP** (PERIOD), **STOP,** or **ENTER,** as shown in black on the individual keys, or to input an appropriate symbol shown in red on the individual keys. The latter are obtained by holding down the **SHIFT** key (lower lefthand corner of keyboard) and then pressing the individual key.

The function cursor F is obtained by holding down the **SHIFT** key and then pressing the **FUNCTION** key. In this mode the functions shown below the individual keys may be selected. After inputting a selected function the microcomputer operating system returns control to the L cursor mode.

If you try to enter commands or data that are not valid in BASIC, the microcomputer operating system which is monitoring your actions will indicate that a mistake has occurred by displaying the syntax cursor, S. To erase the error you can repeatedly use the **DELETE** key (shifted **0**)to remove the part of the line up to and including the S cursor, and then enter the correct BASIC symbols under the re-established L cursor mode.

To enter graphic symbols or valid inverse characters in your BASIC program you must select the GRAPHICS MODE. To do this press the **GRAPHICS** key (shifted 9) and the graphics cursor G will be displayed. This then enables you to enter any number of the 22 graphics symbols (see codes 0 to 10 and 128 to 138 in Table 7-1) or any number of the valid inverse characters (see codes 139 to 191 in Table 7-1). In the graphics mode the required graphics symbol is entered using the appropriate shifted key. To exit from the graphics mode you must press the **GRAPHICS** key again or the **ENTER** key, which re-establishes the L cursor.

To help you familiarize yourself with operating the keyboard we include below a step-by-step description of how to enter a simple three-line BASIC program, which, when run, calculates and displays the square root of a number entered at the keyboard.

Start by switching off the Timex Sinclair 1000/ZX81 9-V dc momentarily to clear the memory and to bring up the K cursor on the screen. Then carry out the following steps and note the display after each step.

Table 2-1.

Your Action	Display After Your Action Lower Left	Upper Left
press 5	5 [K]	none
enter keyword **INPUT**	5 INPUT [L]	none
press X	5 INPUT X [L]	none
press **ENTER**	[K]	5 ■ INPUT X
enter 15	15 [K]	5 ■ INPUT X
enter keyword **PRINT**	15 PRINT [L]	5 ■ INPUT X
enter **FUNCTION** mode (shifted **ENTER**)	15 PRINT [F]	5 ■ INPUT X
enter **SQR** (H key)	15 PRINT SQR [L]	5 ■ INPUT X
press **X**	15 PRINT SQR X [L]	5 ■ INPUT X
press **ENTER**	[K]	5 INPUT X 15 ■ PRINT SQR X
enter 25	25 [K]	5 INPUT X 15 ■ PRINT SQR X
enter keyword **STOP** (shifted A)	25 STOP [L]	5 INPUT X 15 ■ PRINT SQR X
press **ENTER**	[K]	5 INPUT X 15 PRINT SQR X 25 ■ STOP

Remarks or comments can be included in a program by using the **REM** statement. Everything between **REM** and the end of that line is ignored when the Timex Sinclair 1000/ZX81 executes the program. You may wish to verify this by including the line

3 **REM** SQUARE ROOT PROGRAM

in the above program.

You will note that as each line appears in the upper left part of the screen the symbol ■ appears in the line. This is a pointer used in program editing and its use is described in the answer to the next question.

The program has now been entered and can be run. To do this press **RUN** followed by **ENTER.** The program listing is now cleared from the screen and the [L] cursor appears in the lower left corner prompting you to enter the number whose square root you want. Enter the number followed by **ENTER** to execute the program. For example, if you input 64 the result 8 is displayed in the upper left corner and the report 9/25 is displayed in the lower left corner. A glance at Table 2-1 will remind you that this means that a **STOP** statement has been executed at line number 25.

To rerun the program input **RUN** followed by **ENTER.** If you wish to run the program for several different numbers it may be more appropriate to change line 25 to

25 **GOTO** 5

and in this case the program does not stop after displaying the result, but is directed to go back to line 5. The program is now in a continuous loop and will calculate and display the result of 22 entered numbers before stopping and displaying the report code 5/15. To deal with more numbers you should now enter **CONT** followed by **ENTER** to continue.

When you are in the loop and wish to stop execution of the program enter **STOP** followed by **ENTER.** To continue the program after a stop action simply enter **CONT** followed by **ENTER.**

As a further example consider the following one-line program to illustrate use of the graphics mode and the syntax monitoring feature of the Timex Sinclair 1000/ZX81.

Bring up the K cursor as described in the previous example and then carry out the following steps and note the display after each step.

Table 2-2.

Your Action	Display After Your Action	
	Lower Left	Upper Left
press 5	5 K	none
enter keyword **PRINT**	5 PRINT L	none
enter " (shifted P)	5 PRINT " L	none
enter **GRAPHICS** mode (shifted 9)	5 PRINT " G	none
enter ON	5 PRINT " O N G	none
exit from **GRAPHICS** mode (shifted 9)	5 PRINT " O N L	none
enter **ENTER**	5 PRINT " O N L S	none

The S cursor indicates a syntax error because we have omitted the necessary delimiter, ", which must be included at the end of a **PRINT** statement. The correction is made as follows:

Your Action	Display After Your Action Lower Left	Upper Left
enter " (shifted P)	5 PRINT " O N " L	none
enter **ENTER**	K	5 ▶ PRINT " O N "

To run the one-line program press **RUN** followed by **ENTER.** You will now see ON printed in *inverse character form* on the tv screen.

How are programs listed and edited?

When you want to list lines of a program on a display you can use the keyword **LIST.** If you simply input **LIST** and **ENTER** the crt displays the first 22 lines of your program. Alternatively the instruction

LIST line number
ENTER

displays the specified line number and the next 21 lines of program. In this way you can examine any block of 22 lines of your program. Note that the line pointer, ▶, is positioned at the first line listed.

A line in a program can be changed in one of two ways. The first method simply involves entering the whole new line in the normal way, which overwrites the existing line when **ENTER** is pressed. The alternative method is to use the line editing facility. To do this first display the line to be edited, and subsequent lines in the block, by using the instruction

LIST line number to be edited

and then select the **EDIT** mode (shifted 1). The line to be changed now appears in the lower left of the screen, with the K cursor appearing after the line number. Right and left movement of the K or L cursor in the line to be edited is achieved by → (shifted 8) or ← (shifted 5) respectively. The alphanumeric character or word immediately to the left of the cursor may be removed using the **DELETE** key (shifted 0), and an alphanumeric character or word may be inserted at the position of the cursor by entering it from the key-

board. After editing the line it is entered and replaces the original line when **ENTER** is pressed.

To assist editing the line pointer may be moved up or down in a listed block of program by using the ↑ (shifted 7) or ↓ (shifted 6) respectively.

How are the LPRINT, LLIST, and COPY statements used?

When you have a printer you will use these statements to obtain *hard copies,* that is, printouts on paper of programs and results.

The **LPRINT** statement is used in the same way as the **PRINT** statement, differing only in that the output is to the printer instead of to the screen. For example, if you enter

LPRINT "GONE FISHING"

followed by **ENTER**, your message, GONE FISHING, will be printed.

To print lines of your program on the accessory printer use the statement **LLIST.** This is similar in operation to the **LIST** statement used for displaying a maximum of 22 lines of program on the tv screen, but the **LLIST** statement causes *all* lines of your program to be printed. The **LLIST** operation can be terminated by pressing the **BREAK** key. To start printing from a specified line number of a program enter:

LLIST line number

The **COPY** statement is used to obtain a hard copy of the entire screen display. This feature is useful for obtaining printouts of graphs, tables of data, histograms, etc.

How is the INKEY$ function used?

When you *hold* a single key pressed (nonshifted or shifted) the character corresponding to the pressed key is assigned to the string variable **INKEY$**, otherwise, **INKEY$** is a null string (" "). You may verify this by entering and running the following program:

```
 5 IF INKEY$ = " " THEN GOTO 5
15 PRINT INKEY$;
25 GOTO 5
```

You may assign the string variable **INKEY$** to a string variable (a single alphabetic character followed by a $) using the **LET** statement; for example, we may change the above program to the alternative form

```
 5 IF INKEY$ = " " THEN GOTO 5
15 LET K$ = INKEY$
25 PRINT K$;
35 GOTO 5
```

The form and use of strings and string variables is described in Chapter 4.

CHAPTER 3

Which Number Is Which?

Why do I need to know about decimal, binary, and hexadecimal numbers?

We use decimal numbers in our everyday lives and accept and use them readily. Not surprisingly then the data used with BASIC programs is usually in decimal form. The typical program statement

```
120 PRINT 4 * 5.3
```

uses the decimal line number, 120, and the decimal data numbers 4 and 5.3. The Timex Sinclair 1000/ZX81 outputs the answer 21.2 (decimal number) to the screen.

The electronic circuits inside the microcomputer are not designed to handle decimal numbers directly. In fact the decimal numbers used in a program statement are converted by the system into suitable equivalent forms which are acceptable to the electronic circuits. These equivalent forms are the binary representations of the numbers.

We need to know about integer binary numbers when we are concerned with the detailed operation of the microprocessor, especially when we include machine-code instructions in a program. In contrast, it would be unusual to be directly concerned with fractional binary numbers because the Timex Sinclair 1000/ZX81 has floating-point arithmetic capability.

To simplify the handling of binary numbers by the programmer, the more convenient shorthand hexadecimal

notation is used, in which one hexadecimal symbol replaces a group of four binary symbols.

What is a binary number?

The Timex Sinclair 1000/ZX81 microcomputer uses chips (integrated circuits) which have two stable states. These are known as the binary states and they represent two voltage levels, say 0 volts and +5 volts. It is convenient to represent the 0 volt level by the symbol 0 and the +5 volt level by the symbol 1. The symbols 0 and 1 are called *bits* (**b**inary dig**its**).

Numbers can be represented by an arrangement of bits. This is possible because a numeric weighting is given to each bit position within the number. For example, the binary number 10010101 is a shorthand way of writing

$$(1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$
$$= 128 + 0 + 0 + 16 + 0 + 4 + 0 + 1$$
$$= 149$$

The binary number above has a *word-length* of eight bits, and a word of eight bits is called a *byte.* A single byte can represent any integer number in the range zero (00000000) to 255 (11111111). Alternatively a byte can be used to represent a character in the Character Set. For example, the byte 00010101 (21) represents the + character, see Table 7-1.

How do I convert an integer decimal number to its equivalent binary number?

The decimal number is converted by dividing it by 2 to form a quotient and a remainder. The division process is repeated until the quotient is zero, and the remainder formed by the conversion represents the number in binary form. The last remainder is the most significant bit of the binary number. The following example illustrates the method.

You will find a suitable BASIC program in Chapter 8, program 1, which can be used to convert an integer decimal number to its equivalent binary number. We suggest that you run the program yourself to convert several decimal numbers.

	Quotient	Remainder
(Start) 116 ÷ 2 =	58	0
58 ÷ 2 =	29	0
29 ÷ 2 =	14	1
14 ÷ 2 =	7	0
7 ÷ 2 =	3	1
3 ÷ 2 =	1	1
(Finish) 1 ÷ 2 =	0	1
		Read upward

i.e. 116 (decimal) = 1110100 (binary)

How do I convert an integer binary number to its equivalent decimal number?

The most significant bit of the binary number is multiplied by 2 and the next significant bit is added to the result of the product. The result is then multiplied by 2. The next significant bit is now added to the result, and this is multiplied by 2. We continue in this way until all bits of the binary number have been used. The following example illustrates the method:

```
          1    0    1    0    1    1    0    1
(Start) × 2
          2 +  0
               2
             × 2
               4 +  1
                    5
                  × 2
                   10 +  0
                        10
                       × 2
                        20 +  1
                             21
                           × 2
                             42 +  1
                                  43
                                × 2
                                  86 +  0
                                       86
                                     × 2
                                      172 +  1
                                           173  (Finish)
```

i.e. 10101101 (binary) = 173 (decimal)

You will find a suitable BASIC program in Chapter 8, program 2, which can be used to convert an integer binary number to its equivalent decimal number. We suggest that you run the program yourself to convert several binary numbers.

What is a hexadecimal number?

Hexadecimal numbers are a convenient and very useful shorthand representation of their equivalent binary form. This representation uses the numeric symbols 0 to 9 and the alphabetic symbols A to F as shown in the table below

Decimal	Hexadecimal	Binary
0	00	00000000
1	01	00000001
2	02	00000010
3	03	00000011
4	04	00000100
5	05	00000101
6	06	00000110
7	07	00000111
8	08	00001000
9	09	00001001
10	0A	00001010
11	0B	00001011
12	0C	00001100
13	0D	00001101
14	0E	00001110
15	0F	00001111
16	10	00010000
17	11	00010001
18	12	00010010
.	.	.
.	.	.
.	.	.
253	FD	11111101
254	FE	11111110
255	FF	11111111

We can see from the table that an 8-bit binary number is represented by two hexadecimal symbols. The least significant hexadecimal symbol represents the least significant four bits of the binary word, and the most significant hexadecimal symbol represents the most significant four bits of the binary word. A numeric weighting is given to each

hexadecimal symbol; the least significant symbol has a weighting of 1 (16^0) and the most significant symbol has a weighting of 16 (16^1). For example A9 (10101001) may be written as

$$\begin{aligned} & (A \times 16^1) + (9 \times 16^0) \\ = & (10 \times 16) + (9 \times 1) \\ = & 169 \end{aligned}$$

You should note that hexadecimal numbers are not restricted to using just two weighted symbols. A typical example of using four weighted hexadecimal symbols is when a sixteen-bit word (two bytes) is used for the address code of a memory location. The hexadecimal number 62C8 written in binary form is:

0110	0010	1100	1000
6	2	C	8

How do I convert an integer binary number to its equivalent hexadecimal number?

A binary number may be converted directly to its hexadecimal equivalent by starting at the right side of the number and splitting the number into groups of four bits. Zeros are added if necessary to complete a group of four. Each group is then converted directly to the appropriate hexadecimal character. The following example illustrates the method.

zeros
added
↓↓

$$\underbrace{0011}_{3} \quad \underbrace{0111}_{7} \quad \underbrace{1011}_{B} \quad \underbrace{1100}_{C}$$

i.e. 11011110111100 (binary) = 37BC (hex)

How do I convert an integer hexadecimal number to its equivalent binary number?

A hexadecimal number may be converted directly to its binary equivalent by converting each symbol to its 4-bit binary equivalent, ensuring that the positional arrangement

is maintained. The following example illustrates the method:

$$\underbrace{E}_{1110}\quad\underbrace{2}_{0010}\quad\underbrace{6}_{0110}\quad\underbrace{C}_{1100}$$

i.e. E26C (hex) = 1110001001101100 (binary)

How do I convert an integer decimal number to its equivalent hexadecimal number?

The decimal number is converted by dividing it by 16 to form a quotient and a remainder. The division process is repeated until the quotient is zero, and the remainders formed by the conversion represent the number in hexadecimal form. The last remainder is the most significant digit of the hexadecimal number. The following example illustrates the method:

	Quotient	Remainder
(Start) 9886 ÷ 16	617	14 (E) ↑
617 ÷ 16	38	9
38 ÷ 16	2	6
(Finish) 2 ÷ 16	0	2
		Read upward

i.e. 9886 (decimal) = 269E (hex)

You will find a suitable BASIC program in Chapter 8, program 1, which can be used to convert an integer decimal number to its equivalent hexadecimal number. We suggest that you run the program yourself to convert several decimal numbers.

How do I convert an integer hexadecimal number to its equivalent decimal number?

The decimal equivalent of the most significant symbol of the hexadecimal number is multiplied by 16, and then the decimal equivalent of the next symbol in the hexadecimal number is added to the result of the product. The result is multiplied by 16. The decimal equivalent of the next symbol in the hexadecimal number is now added to the result, and this is multiplied by 16. We continue in this way until all

symbols of the hexadecimal number have been used. The following example illustrates the method.

1	0	F	A	
		replace by decimal equivalent	replace by decimal equivalent	
× 16				
(Start) 16 +	0			
	16			
	× 16			
	256 +	15		
		271		
		× 16		
		4336 +	10	
			4346	(Finish)

i.e. 10FA (hex) = 4346 (decimal)

You will find a BASIC program in Chapter 8, program 2, which can be used to convert an integer hexadecimal number to its equivalent decimal number. We suggest that you run the program yourself to convert several hexadecimal numbers.

What is binary arithmetic?

The Z80A microprocessor performs arithmetic operations using binary representations of numbers, and consequently a basic understanding of the concepts of binary arithmetic is desirable. To perform binary arithmetic operations is simply a matter of knowing and applying the basic simple rules for addition, subtraction, multiplication and division.

The basic rules of binary addition are

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 0 \text{ with a carry } 1$$
$$1 + 1 + 1 = 1 \text{ with a carry } 1$$

Two examples of the addition of two 4-bit binary numbers, with their equivalent decimal values, are shown below.

Decimal	Binary	Decimal	Binary
9	1001	7	0111
+ 6	+0110	+ 4	+0100
15	1111	11	1011

Binary subtraction may be performed using the basic binary subtraction rules, which are:

$$\begin{aligned} 0 - 0 &= 0 \\ 1 - 0 &= 1 \\ 1 - 1 &= 0 \\ 0 - 1 &= 1 \text{ with a borrow } 1 \\ 1 - 1 - 1 &= 1 \text{ with a borrow } 1 \end{aligned}$$

Two examples of the subtraction of two 4-bit binary numbers, with their equivalent decimal values, are shown below:

Decimal	Binary	Decimal	Binary
12	1100	10	1010
− 2	0010	− 9	1001
10	1010	1	0001

In the two subtraction examples above the results are positive numbers, but the result of a subtraction may be positive or negative depending on the relative magnitudes of the minuend and subtrahend. In order to decide whether a number is positive or negative the leftmost bit (most significant bit, MSB) is used as a *sign bit.* A commonly used convention is that the sign bit is 1 if the number is negative and 0 if the number is positive. Consequently the Z80A microprocessor, which has eight bits available for data, uses seven bits for the magnitude of the data and the most significant bit to indicate the sign.

Complex electronic circuitry is required to subtract binary numbers directly, and in microcomputers it is usual to perform subtractions by adding the two's complement of the subtrahend to the minuend. This means that subtractor circuitry is not required: subtraction is performed using adder circuits to implement the complement form of arithmetic operations.

The two's complement of a binary number is found by converting each bit of the binary number, that is, by changing all the 0s to 1s and 1s to 0s, and adding 1 to the result. For example, the two's complement of 12 using an 8-bit binary representation is obtained as follows:

sign bit

+ 12 = 00001100

change 1s to 0s and 0s to 1s

11110011

+ 1 add 1

11110100 = − 12 in two's complement form

To perform subtraction using the two's complement method, the two's complement of the subtrahend must be obtained, and this is then added to the minuend. As an example let us consider subtracting decimal 40 from decimal 31 using 8-bit binary words.

sign bit sign bit

31 = 00011111, 40 = 00101000,

sign bit

hence: − 40 in two's complement form is 11011000,

sign bit

	0	0011111	
+	1	1011000	
	1	1110111	= − 9 in two's complement form

In this case the sign bit indicates a negative number which is in two's complement form.

As a further example of the two's complement method of subtraction we shall consider subtracting the decimal fraction 0.25 from the decimal fraction 0.875 using 8-bit binary words. In this case we must convert both decimal fractions to their equivalent binary representations. This is done as follows:

$$\begin{aligned} 0.875 &= 0.5 + 0.25 + 0.125 \\ &= (1 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) \end{aligned}$$

In shorthand notation we write

$$0.875 = 0.111$$

Similarly, $0.25 = (0 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3})$

i.e., $0.25 = 0.010$.

Using 8-bit binary words for the binary numbers, with the binary point occurring after the fourth bit, we have

```
      sign bit  binary point    sign bit  binary point
          ↓       ↓                  ↓       ↓
0.875 = 0000.1110,           0.25 = 0000.0100
```

```
                                          sign bit  binary point
                                                 ↓   ↓
hence: −0.25 in two's complement form is 1111.1100
```

```
                 sign bit    binary point
                     ↓           ↓
                     0        000.1110
                   + 1        111.1100
   ignore          ---------------------
   overflow  →       1 0      000.1010  = 0.625
                   ---------------------
```

A common method of binary multiplication uses the shift and add principle. The multiplicand is multiplied by each bit of the multiplier on a bit-by-bit basis and the resultant product is obtained by adding all the appropriately shifted partial products. As the multiplier bits are either 0 or 1 the partial product terms are either zero or equal to the multiplicand or shifted versions of the multiplicand.

As an illustration consider the multiplication of the decimal numbers 43 and 11. We use a 16-bit accumulator to hold the result and work with 8-bit binary number representations for the multiplicand and the multiplier.

```
Multiplicand 00101011 = 43
  Multiplier 00001011 = 11
             --------
             00101011 \  Appropriately shifted
            00101011   } partial product terms
          00101011    /
    ----------------
    0000000111011001     Sum of partial product terms = 473
    ----------------
```

When two binary numbers are multiplied the product contains a number of bits equal to the sum of the bits contained

in the two binary numbers. The maximum number of additions required for a full multiplication using the shift and add method is equal to the number of bits in the multiplier.

Binary division can be performed by using a shift and subtract method. It is implemented by successively subtracting the divisor from the appropriate shifted dividend, and inspecting the sign of the remainder after each subtraction. If the sign of the remainder is positive the value for the quotient is 1, but if the sign is negative the value is 0, and the dividend is restored to its previous value by adding back the divisor (restoring). After the subtraction yielding a positive quotient, or after the restoration following a negative quotient, the divisor is shifted one place to the right and the next significant bit of the dividend is included and the operation repeated until all bits in the dividend have been used.

We illustrate the method by considering the division of 90 by 9 using corresponding 8-bit binary number representations.

What is floating point number representation and why is it used?

When using BASIC in the Timex Sinclair 1000/ZX81, decimal numbers are represented using floating-point binary notation. This permits manipulation of decimal numbers in the range $\pm 1.7 * 10^{38}$ using at least eight significant decimal digits.

In floating point form decimal numbers are represented as:

$$\text{Number} = m * 2^{\varepsilon}$$

where ε is the exponent and m is the mantissa. The range of the mantissa is restricted to

$$\tfrac{1}{2} \leqslant m < 1$$

and the exponent ε is an integer in the range

$$-127 \leqslant \varepsilon \leqslant +127$$

The mantissa is stored as a 4-byte fractional binary number, and because of its restricted range the most significant bit is always 1. This means that the most significant

00001010	
$00001001\overline{)\ 01011010}$	
-00001001	
11110111	Result negative; quotient = 0 Add divisor back
$+00001001$	
000000001	
-00001001	Shift divisor right and subtract
11111000	Result negative; quotient = 0 Add divisor back
$+00001001$	
000000010	
-00001001	Shift divisor right and subtract
11111001	Result negative; quotient = 0 Add divisor back
$+00001001$	
000000101	
-00001001	Shift divisor right and subtract
11111100	Result negative; quotient = 0 Add divisor back
$+00001001$	
000001011	
-00001001	Shift divisor right and subtract
000000100	Result positive; quotient = 1
-00001001	Shift divisor right and subtract
11111011	Result negative; quotient = 0 Add divisor back
$+00001001$	
000001001	
-00001001	Shift divisor right and subtract
000000000	Result positive; quotient = 1
-00001001	Shift divisor right and subtract
11110111	Result negative; quotient = 0 Add divisor back Remainder
$+00001001$	
00000000	

bit of the mantissa is redundant and it is therefore possible to use it to indicate the sign of the number. The notation adopted for this sign bit is to use 0 for positive numbers and 1 for negative numbers.

The exponent is stored using one byte. To avoid unnecessary complication regarding the sign of the exponent it is stored as a positive integer number in the range 1 to 255, achieved by adding the decimal number 128 to the actual exponent value, ε.

The computer represents zero by setting the four bytes of the mantissa and the exponent byte to 0.

Using the floating point format the maximum positive number that can be handled corresponds to when all bits in the exponent byte are set to 1 representing $\varepsilon = +127$, and all 32 bits in the 4 bytes of the mantissa are also set to 1 representing $m = (1 - (1/2^{32}))$. The maximum possible number is therefore

$$= 2^{127} \times (1 - (1/2^{32}))$$
$$\cong 2^{127} \text{because } (1 - 1/2^{32}) \cong 1$$
$$\cong 1.7014 \times 10^{38}$$

We can confirm that the machine can handle this number by entering **PRINT** 1.7014E + 38 in which case the computer responds by displaying 1.7014E + 38, indicating that the number is acceptable. However, if we increase this number to say 1.7015×10^{38}, which we enter as **PRINT** 1.7015E + 38, the computer will not accept the number because it is out of range.

The smallest positive number that can be represented in the computer corresponds to when the exponent byte produces $\varepsilon = -127$, and the mantissa value is 0.5. The minimum possible number is therefore

$$= 2^{-127} \times 0.5$$
$$= 2^{-128}$$
$$\cong 2.9388 \times 10^{-39}$$

We can check that the ZX81 can handle this number by entering **PRINT** 2.9388E – 39 in which case the computer responds by displaying 2.9388E – 39 indicating that the number is acceptable. If however we try to input a smaller number, say 2.92E – 39, by inputting **PRINT** 2.92E – 39, the computer will not accept it but outputs the smallest positive

number 2.9387359E – 39. For numbers much smaller than 2.9387359E – 39 the Timex Sinclair 1000/ZX81 approximates the value to zero.

The Timex Sinclair 1000/ZX81 uses a sign and magnitude format for number representation which means that the positive and negative number ranges are identical.

A simple BASIC program to investigate the acceptable number range of the ZX81 is listed below.

```
10 INPUT X
15 CLS
20 PRINT X
30 PRINT
40 INPUT Y
50 PRINT Y
60 PRINT
70 PRINT X + Y
80 GOTO 10
```

You can now input numerical data values for X and Y, and their sum is formed and then displayed. If X and Y are within the acceptable number range their values will be displayed, but their sum is only displayed if it too is within the acceptable number range.

To exit from the program simply press the **STOP** Key (shifted A).

How do I specify a memory address, and how do I examine and change the number stored in an addressed memory location?

The Z80A microprocessor uses a 16-bit binary word to address a location in memory. However it is more convenient to use the decimal equivalent of the 16-bit address word, and consequently the BASIC interpreter in the Timex Sinclair 1000/ZX81 allows you to refer to a memory address using a decimal number.

To examine the stored content of any memory location in ROM or RAM, we use the **PEEK** function and the address of the memory location in decimal form. For example

PRINT PEEK 561

will output to the screen the content of the Read Only

Memory location 561, which has the fixed stored value 124. You may wish to try this for yourself to check the result.

It is only possible to change the content of a Random Access Memory location. We can do this using the **POKE** statement with the address of the memory location and the data in decimal form. For example

POKE17111, 123

will load the 8-bit binary data word 01111011 (123 decimal) into memory location 0100001011010111 (17111 decimal), which makes clear why decimal number representation is more convenient than the equivalent binary representation. You can of course check that the data has been stored by **PEEK**ing this memory location:

PRINT PEEK 17111

You can store the positive integer numbers in the range 0 to 255 using the **POKE** statement. If you attempt to store integer numbers greater than 255 the computer responds with the message B/0, indicating that the integer number is out of range.

You should note that if you **POKE** negative integer values in the range −1 to −255 the computer responds with 0/0 thereby indicating acceptance of the negative integer number. However, it is important to realize that negative integers are stored in complement form. This means, for example, that if you store −123 it is actually stored as 133. Note that 133 = 256 − 123, which implies that the stored integer value is obtained by subtracting the magnitude of the entered negative number from 256.

If you try to store a noninteger number it is rounded to the nearest integer value and then stored. For example try **POKE**ing 4.73 into memory location 17111. You may **PEEK** this location and see that the number is stored as 5. If you try to store 5.5 you will find that it is stored as 6, whereas 5.4999 is stored as 5. A negative noninteger number is rounded and stored in complement form. For example −13.417 is stored as 243 (i.e., 256 − 13).

CHAPTER 4

Number Crunching and Pulling Strings

What is a numeric variable?

To use the Timex Sinclair 1000/ZX81 to sum three numbers, say 2.4, 4.1, and 5.2, we could simply command the computer to

PRINT 2.4 + 4.1 + 5.2

and the computer would display 11.7.

On the other hand if we wish the computer to add *any* three numbers then we could assign three different symbols to represent the three numbers, and use these symbols in a BASIC program. For example, a simple program to obtain the required sum using the symbols A, B and C for the three numbers, is

```
10 INPUT A
20 INPUT B
30 INPUT C
40 PRINT A + B + C
```

The symbols A, B, and C can have any valid numeric value and consequently they are called *numeric variables.* A numeric variable can be named using alphabetic characters, numeric characters and spaces, but the first character must be alphabetic. Some valid numeric variables' names are

TREVOR
FISH
BOB 1
THE GRAND TOTAL IS

However, the following numeric variables' names are not allowed

3 SUE	because it begins with a numeric character
A$10	because it uses an invalid character ($)
FISH "N CHIPS	because it uses an invalid character (")

There are two BASIC statements which are often used with numeric variables. These are the **LET** and **CLEAR** statements. The **LET** statement can be used to assign a constant value to a variable, for example

LET TROUT = 8

or it is used to assign the value of the right side of a mathematical expression, involving already defined numeric variables, to a new numeric variable. Typical examples are

LET X = 2 * Y + C
LET SUM 1 = −(4.9/T + A)

The **CLEAR** statement eliminates all defined variables and frees all of the memory that had been assigned to the variables.

How do I use the computer to perform simple arithmetic calculations?

The computer can directly perform five arithmetic operations, namely addition (+), subtraction (−), multiplication (*), division (/) and exponentiation, or raising to a power (**). When several arithmetic operations are used in the same calculation it first calculates any exponentiations, then any multiplications and divisions and finally any additions and subtractions. However, if calculations are placed in parentheses they are evaluated first and separately.

The best way to appreciate how calculations are worked out on the ZX81 is to consider some examples. Try these for yourself.

1. Required calculation: $12 \div 3 + 6^2$
   Computer statement: **PRINT** 12/3 + 6**2
   Displayed answer: 40

2. Required calculation: $12 \div (3 + 6^2)$
   Computer statement: **PRINT** 12/(3 + 6**2)
   Displayed answer: 0.30769231
3. Required calculation: $((5.7 \times 32) - 7.6) \times (-2.1)$
   Computer statement: **PRINT** ((5.7*3.2) − 7.6)*(−2.1)
   Displayed answer: −22.344
4. Required calculation: 6.3^{21}
   Computer statement: **PRINT** 6.3**21
   Displayed answer: 6.1115524E + 16
5. Required calculation: 14.2^{-20}
   Computer statement: **PRINT** 14.2** −20
   Displayed answer: 8.9998052E − 24

Note that the displayed answers to examples 4 and 5 are given in scientific notation form, where the E + 16 represents 10^{16} and E − 24 represents 10^{-24}. This form of notation is a shorthand method of representing very large, or very small numbers, and is sometimes referred to as floating-point representation.

We can include numeric variables in arithmetic expressions and the computer will then use their respective numeric values in the calculation. Suppose that we wish to write a program to calculate the average of any three numbers, then we can use the numeric variables A, B and C to represent the numbers, and calculate the average using the program listed below. Try it on your machine.

```
15 INPUT A
25 INPUT B
35 INPUT C
45 PRINT (A + B + C)/3
```

As a further example here is a program that will calculate both the square and the square root of a number:

```
15 INPUT A
25 PRINT A**2,A**.5
35 PRINT A**2;A**.5
45 PRINT A**2;" ";A**.5
```

We suggest that you try this program and note the three different displayed answer formats created by the three different **PRINT** statements.

What is a numeric variable array?

If several variables have one common characteristic, for example Cost, it is more convenient to name the variables by using a symbol for the common characteristic, and to distinguish between the variables by using an identification number. For example, we may wish to record weekly expenditure on (a) bus fares, (b) sweets, (c) lunches, (d) drinks, and (e) pet food. We require five variables for these items, and since the common characteristic is Cost, we can denote them as C(1), C(2), C(3), C(4) and C(5), as shown in Fig. 4-1.

	BUS FARES	SWEETS	LUNCHES	DRINKS	PET FOOD
YOU	C(1)	C(2)	C(3)	C(4)	C(5)

Fig. 4-1. One-dimensional array with five elements.

The arrangement shown in Fig. 4-1 is known as a one-dimensional array containing five elements. To use this array in a BASIC program it is necessary first to reserve the required memory locations for the array by using the **DIM** statement. In this example, as there are five elements in this one-dimensional array, the appropriate **DIM** statement is **DIM** C(5). It is important to note that the number distinguishing the variables is enclosed in parentheses. This means that C(1), C(2), C(3), C(4) and C(5) are the uniquely named valid numeric variables. You may only use parentheses in a variable name when dealing with arrays.

The simple BASIC program listed below requires as input the cost of the five items, and then it calculates the total cost.

```
10 DIM C(5)
20 INPUT C(1)
30 INPUT C(2)
40 INPUT C(3)
50 INPUT C(4)
60 INPUT C(5)
70 LET T = C(1) + C(2) + C(3) + C(4) + C(5)
80 PRINT "TOTAL WEEKLY COSTS IS £"; T
```

Note that the above program has five separate input statements and in programming terms this is rather inefficient. An alternative and more general BASIC program which can handle N separate positive input values, where N is any acceptable positive integer, is listed in Chapter 8, Program 3. Try that program yourself with various sets of data.

We are not restricted to using arrays with just one dimension. For example, it is possible to extend the one-dimensional array considered above to a two-dimensional form to include cost-analysis data for the expenditure incurred by several people. In Fig. 4-2 we show a two-dimensional array representing five cost items for four people, yourself and three friends. This array contains 20 elements arranged in four rows and five columns. Each element is identified as a subscripted variable using the format:

C (row number, column number).

	BUS FARES	SWEETS	LUNCHES	DRINKS	PET FOOD
YOU	C(1,1)	C(1,2)	C(1,3)	C(1,4)	C(1,5)
Friend A	C(2,1)	C(2,2)	C(2,3)	C(2,4)	C(2,5)
Friend B	C(3,1)	C(3,2)	C(3,3)	C(3,4)	C(3,5)
Friend C	C(4,1)	C(4,2)	C(4,3)	C(4,4)	C(4,5)

Fig. 4-2. Two-dimensional array with four rows and five columns.

For example, C(3, 4) refers to the cost of drinks for Friend B.

To use this array in a BASIC program we must, of course, reserve the necessary memory locations by using at the head of the program:

DIM C(4, 5)

A BASIC program, which accepts data for the 20-element, two-dimensional array, and outputs the total expenditure

per person and the total expenditure by the four people for each cost-item, is listed in Chapter 8, Program 4. We suggest that you run the program yourself with various sets of data.

Theoretically, there is no restriction on the array size or the number of array dimensions, but in practice you will find that the storage capacity of your machine's memory is the limiting factor. Remember that the number of memory locations reserved is the *product* of the total number of elements in each dimension.

What is a string and how is it used?

When you wish to include text in your program you can use the **PRINT** statement followed by the text inserted in quotation marks. The text, or string of characters, is referred to as a string. For example "HAPPY BIRTHDAY" is a string and this can be displayed on the screen by using

PRINT "HAPPY BIRTHDAY"

In a string you can use all the available characters on the keyboard, except the quotation mark because this will be interpreted as the end of the string. If you wish to include quotation marks *within* a string you must use the double quotation-mark key (shifted Q) and this will be printed as a single quotation mark within the string. For example,

PRINT "HAPPY BIRTHDAY""MOM"""

will display HAPPY BIRTHDAY "MOM" on the screen.

You can obtain a count of the LENgth of a string by using the **LEN** function. For example, the string "HAPPY BIRTHDAY" ""MOM""" has 19 characters and you can confirm this by typing

PRINT LEN "HAPPY BIRTHDAY""MOM"""

and you will note that the machine responds by displaying 19 on the screen.

When a string contains no characters between the quotation marks, it is known as a *null string.*

Strings can be added together using the + operation. For example, try the following

PRINT "HAPPY BIRTHDAY" + "DAD"

But note that only the + operation can be used in this way.

You may assign a string to a variable and to do this a single alphabetic character followed by a $ is used to denote the variable. Therefore you can only have a maximum of 26 string variables in your BASIC program. The following simple program uses five string variables:

```
15 LET A$ = "HAPPY BIRTHDAY"
25 LET B$ = " "
35 LET C$ = "DAD"
45 LET D$ = "MOM"
55 LET K$ = A$ + B$ + C$
65 PRINT K$
```

Verify that the program displays HAPPY BIRTHDAY DAD. Try changing the string variable C$ to D$ in line 55 and rerun the program.

You will find a BASIC program in Chapter 8, Program 5, which uses string variables to create a limited dictionary of words which are concatenated in several ways to display messages. We suggest that you run the program to display the messages and then respond to the displayed instruction.

You can eVALuate strings that are arithmetic expressions by using the **VAL** function. For example

PRINT VAL "5**5**5"

evaluates $((5)^5)^5 = 5^{25} = 2.9802323E + 17$. It is important to note that when the **VAL** function is used with any of the graphic statements **PRINT AT, PLOT,** and **UNPLOT,** it must be the x-coordinate in the statement. For example, the statement

PRINT AT VAL "3**2 + 2",16;"$"

displays $ in the center of the screen (11 rows down and 16 columns across).

The **STR$** function permits you to convert decimal numbers into strings. For example, the statement

PRINT STR$ 56

displays 56 on the screen. It is equivalent to using the statement

PRINT "56"

What is a substring?

A substring is any set of consecutive characters extracted from an existing string. To define a substring we use the string or string variable with the **TO** statement in the format:

String
or   (Start **TO** Stop)
String variable

Therefore if the existing string is to be "CHARLIES ANGELS", we can define this as the string variable A$:

15 **LET** A$ = "CHARLIES ANGELS"

We can then extract and display the substring "AN" by using

25 **PRINT** A$ (10 **TO** 11)

To display the substring "A" we can use

35 **PRINT** A$ (10 **TO** 10)

If you omit the Start position in defining the substring, it is assumed that you wish to start the substring at the first position in the string. So to display "CHAR" we can use

36 **PRINT** A$ (**TO** 4)

Similarly if the Stop position of the substring is omitted it is assumed that the last position in the string defines the end of the substring. For example, to display "ANGELS" we can use

46 **PRINT** A$ (9 **TO**)

When the substring Start position is greater than the Stop position a null substring is created.

Note that the Stop position of the substring must not exceed the position of the last character in the existing string. Furthermore you cannot use negative numbers to define the Start and Stop positions of a substring.

Can we have string arrays?

Yes, it is possible to set up an array of string characters. A one-dimensional array having N elements, where N is a positive integer, must be DIMensioned using a single alpha-

betic character followed by $ and the dimensions of the array. For example:

DIM D$(1,N)

Each element is identified as a subscripted string variable using the format:

D$(1,element number)

see Fig. 4-3.

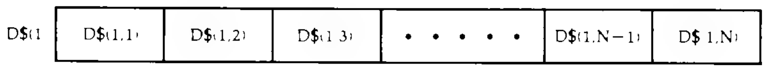


Fig. 4-3. One-dimensional string array.

Consider the one-dimensional string "TELEVISION" which contains 10 characters. To define this string we can use the following:

15 **DIM** D$(1,10)
25 **LET** D$(1) = "TELEVISION"

and this is assigned in the array as shown in Fig. 4-4. Therefore to print a single character from the string, say V, we could use

PRINT D$(1,5)

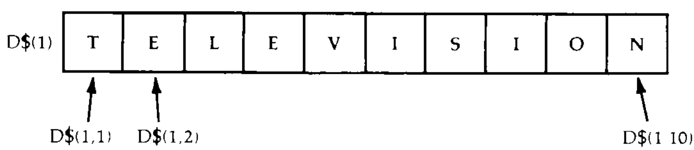


Fig. 4-4. One-dimensional string array for a string containing 10 characters.

Furthermore, to print on the screen the substring LEV we would use the statement

PRINT D$(1,3 **TO** 5)

while to print the entire string we can use the statement

PRINT D$(1)
or **PRINT** D$(1, **TO**)

Using this approach we can refer to individual characters within the string, to substrings, or to the entire string.

A two-dimensional array is constructed in a similar way. However, the DIMension statement must reserve sufficient memory for the longest string. For example, if we have the three strings "JANUARY", "JUNE", and "MAY" we can dimension the array by using

DIM M$(3,7)

where 3 is the number of strings and 7 is the number of characters in the longest string. The arrangement of the corresponding two-dimensional array is shown in Fig. 4-5. Note how much storage space is unused.

M$(1)	J	A	N	U	A	R	Y
M$(2)	J	U	N	E			
M$(3)	M	A	Y				

Fig. 4-5. Two-dimensional string array defining three strings.

It is worth noting that if we try to display the substring by

PRINT M$(2,3 **TO** 6)

we display NE, since elements M$(2,5) and M$(2,6) are null.

How do I use the mathematical functions: ABS, SGN, INT, SQR, LN, and EXP?

The first point to note is that, in the absence of parentheses, these functions are evaluated before calculations in an expression involving the simple arithmetic operations + – * / and **. The rule that expressions enclosed in parentheses are evaluated first applies even when the total expression contains one or more mathematical functions.

The **ABS** function is used when you require the **ABS**olute value of either a number or an evaluated mathematical expression. The term "absolute value" refers only to the magnitude of the number or evaluated expression, with the

sign ignored. Use your ZX81 to obtain the results of the following

```
15 PRINT ABS 118.97, ABS - 118.97
25 PRINT ABS 2.9/1.12*2.81, ABS 2.9/-1.12*2.81
```

Did you notice that the fourth result is prefixed by the − sign? The reason for this is that, because the fourth expression does not contain parentheses, the machine first works out the absolute value of 2.9 and then divides this by −1.12*2.81 (= −3.1472).

The **SGN** function is used to determine whether a number, or an evaluated mathematical expression, is zero, positive or negative. The result obtained when using this function is 0, +1 or −1 corresponding to the zero, positive and negative states respectively. A typical application of this function is where we wish the ZX81 to accept only entered positive numbers greater than zero. This can be demonstrated using

```
 5 PRINT "INPUT POS. NO. > 0"
15 INPUT N
25 LET A = SGN N
35 IF A = 0 THEN GOTO 5
45 IF A = -1 THEN GOTO 5
55 PRINT N;" ENTER NEXT POS. NO"
65 GOTO 15
```

This program takes in a number and, if it is positive, it displays the number and asks you to enter your next number. A negative number or zero is not accepted and you are instructed to enter a positive number greater than zero.

The **INT** function is used to round down any number to the nearest integer value. A positive number is therefore reduced in magnitude by using this function (unless the number is already an integer) while a negative number is increased in magnitude (unless it too is already an integer). For example, +13.732 is reduced to +13 but −13.7321 becomes −14, i.e. its magnitude is increased. Here is a simple program which you can use to verify this for yourself

```
 5 PRINT "INPUT A NUMBER"
15 INPUT A
25 LET B = INT A
```

```
35 PRINT "THE INTEGER OF THE NUMBER IS ";B
45 GOTO 5
```

To obtain the square root of a number you can use the **SQR** function. Try the following program which has been written to find the square root of a positive number. You will note that we have used the **SGN** function to ensure that only positive numbers are accepted by the program. This has been included in the program because the Timex Sinclair 1000/ZX81 cannot calculate the square root of a negative number.

```
 5 PRINT "INPUT POS. NO. > 0"
15 INPUT N
25 LET A = SGN N
35 IF A = 0 THEN GOTO 5
45 IF A = -1 THEN GOTO 5
55 LET B = SQR N
65 PRINT "THE SQUARE ROOT IS =";B
75 GOTO 5
```

To find the natural logarithm of a number we have the **LN** function. The natural logarithm of a number, or $\log_e$ as it is sometimes called, is related to $\log_{10}$ by the relationship

$$\log_{10}x = \log_e x/2.302585093$$

So we can use this to obtain $\log_{10}$ of a number (X). The program listed below uses the **LN** function and the above relationship to evaluate $\log_{10}$ of a positive number.

```
 5 PRINT "INPUT POS. NO. > 0"
15 INPUT N
25 LET A = SGN N
35 IF A = 0 THEN GOTO 5
45 IF A = -1 THEN GOTO 5
55 LET X = LN N/2.302585093
65 PRINT "THE LOG 10 IS = ";X
75 GOTO 5
```

We may, of course, find the antilog_{10} of a number by using the relationship.

$$y = \text{antilog}_{10}x = 10^x$$

For example, in BASIC:

```
125 LET Y = 10**X
```

For a number x, we may evaluate the exponential function e^x, using the **EXP** function. This relationship can be used, for example, to evaluate exponential growth or decay. The program below illustrates the use of this function to calculate the exponential growth of the mathematical term e^{3t}, for t = 0, 1, 2, . . . 10.

```
 5 FOR T = 0 TO 10
15 LET A = EXP (3*T)
25 PRINT A
35 NEXT T
```

We suggest that you run the program and then change line 15 to

```
15 LET A = EXP (-3*T)
```

Now rerun the program and observe the corresponding exponential decay.

How do I find the sine, cosine, and tangent of an angle?

The ZX81 can determine the sine, cosine and tangent of an angle using the **SIN, COS** and **TAN** functions respectively. However, the angle must be expressed in radians, so if the angle is expressed in degrees use the relationship

Angle (in radians) = Angle (in degrees) × $\pi/180$

To implement this on your ZX81 use the **PI** function in the equation.

The program below calculates the sine, cosine and tangent of an angle in degrees.

```
15 PRINT "INPUT ANGLE IN DEGREES"
25 INPUT A
35 LET B = A*PI/180
45 PRINT "SIN A ="; SIN B, "COS A ="; COS B,
   "TAN A ="; TAN B
55 GOTO 15
```

Try this program for various positive and negative angles. You will note that you cannot obtain the tangent of 90 degrees or 270 degrees because, of course, the value is infinite and outside the number range of the computer.

How do I find the value of an angle in degrees if I know its sine or cosine or tangent?

If you know the sine of the angle you can find that angle by using the arcsine function **ASN.** Similarly, if you know the cosine of the angle you find the angle with the arccosine function **ACS,** and given the tangent of the angle you can find the angle by using the arctangent function **ATN.**

These three functions give the resultant angle in radians and therefore to convert to degrees we need to use the relationship

Angle (degrees) = Angle (radians) × $180/\pi$

The following program calculates the angle in degrees for an entered sine value:

```
15 PRINT "INPUT SINE VALUE"
25 INPUT S
35 LET A = ASN S
45 LET B = A*180/PI
55 PRINT "ANGLE IS ";B;" DEGREES"
65 GOTO 15
```

We suggest you run this program for positive and negative arcsine values in the valid range +1 to −1, and you will note that it displays the calculated angle in the range +90 degrees to −90 degrees.

If you change line 15 to

```
15 PRINT "INPUT COSINE VALUE"
```

and line 35 to

```
35 LET A = ACS S
```

the program above can be used to calculate the angle in degrees for an entered cosine value. Note that on running the program it displays the calculated angle in the range 0 degrees to 180 degrees corresponding to valid arccosine input values in the range +1 to −1.

If you change line 15 to

```
15 PRINT "INPUT TANGENT VALUE"
```

and line 35 to

```
35 LET A = ATN S
```

the program can be used to calculate the angle in degrees for an entered tangent value. In this case you will see that the program displays the calculated angle in the range +90

degrees to −90 degrees. Try the arctangent program for input values in the range +90,000 to −90,000.

Can I generate numbers which appear random?

You can use the **RAND** statement with the **RND** function to generate numbers which have some degree of randomness, although they would not strictly satisfy a rigorous mathematical analysis for statistical randomness.

The **RND** function generates a defined sequence of 65535 numbers in the range 0.0022735596 to 0.99885559, and the associated **RAND** statement:

RAND X

where X is a positive integer in the range 1 to 65535, defines the starting point in the **RND** number sequence. However, for randomness it is desirable to have the **RND** function starting off at random points in its number sequence. This is achieved by omitting X or setting it to 0 in the **RAND** statement.

The following program can be used to generate positive random numbers (well, at least they seem random). The program requires you to input the maximum possible magnitude of a generated number. When you have done this, twenty-two numbers of the sequence are generated and displayed.

```
 5 PRINT "INPUT MAX RANDOM NO."
10 INPUT N
15 RAND
25 LET A = 1E8*RND
35 LET B = INT A
45 LET Y = INT ((N + 1)*(A - B))
55 PRINT Y
65 GOTO 15
```

We suggest you run the program yourself. Do you think that the generated numbers are random? You might like to investigate the effect on the randomness of changing line 15 to include a valid positive integer after the **RAND** statement.

You will find a BASIC program in Chapter 8, Program 6, which uses the random feature to generate die numbers which you can use for board-games.

CHAPTER 5

Rolf's Watch

What is a flowchart?

The first consideration in preparing a program is to establish the logical sequence of steps to be performed by the microcomputer to achieve the desired result. This initial preparation may involve drawing a *flowchart* which illustrates diagrammatically the program steps.

A flowchart is made up of graphical symbols which are connected together by straight lines. Arrowheads are drawn on the connecting lines to indicate the direction of flow in the program. A selection of commonly used flowchart symbols is shown in Fig. 5-1.

The start and end of the flowchart is indicated by the *terminal symbol* (Fig. 5-1A) and obviously it is connected to the flowchart by only one line. The *rhomboid symbol* (Fig. 5-1B) is used to show a specific operation by an input or output device. The *rectangle symbol* (Fig. 5-1C) is used to indicate that a specific action is to be taken. A statement inside the rectangle specifies the action, and this may be stated in plain English, or alternatively it could be a logical or algebraic expression. The *diamond symbol* (Fig. 5-1D) is used to indicate the point in a program at which a decision has to be made. For example, it may be necessary at some point in the program to know whether or not the numeric variable A is greater than zero (see Fig. 5-1D) and obviously the answer to this question is either no or yes. Clearly, the

(A) *Terminal symbol.*

(B) *Input/output symbol.*

(C) *Action symbol.*

(D) *Decision symbol.*

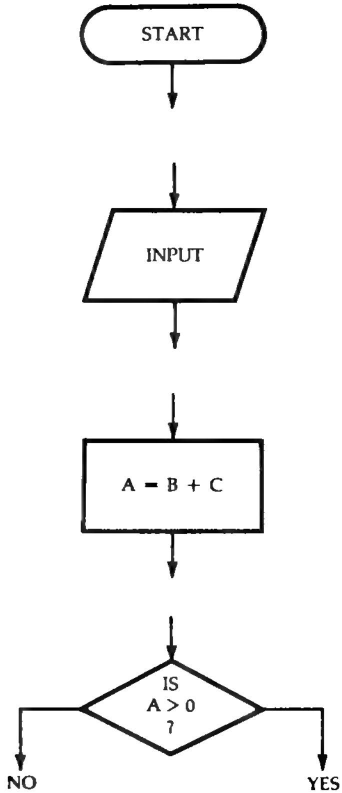


Fig. 5-1. Flowchart symbols.

program will proceed along one of two possible paths depending on the decision made.

As an example Fig. 5-2 shows the basic flowchart that can be used to represent the generation of a time delay. Note that the delay duration is dependent on the values assigned to the positive integer numeric variables X and Y. We suggest you study this flowchart to understand the significance of each step in the sequence of operations. You will probably find it convenient to assign values to X and Y, say 3 and 2 respectively, and to work out with pencil and paper the intermediate values of X and Y after each step in the program. Note that by changing the values of X and Y, the number of steps in the program is correspondingly

changed; hence the time required to execute the program can be varied; and consequently from Start to Stop there is a variable time delay.

How do I translate a flowchart into a program?

For each step in the logical sequence of the flowchart we can select appropriate BASIC functions or statements which can be used to implement the desired operation.

To illustrate the principles involved let us write a time delay program corresponding to the flowchart shown in Fig. 5-2. The steps in the program are tabulated below:

Flowchart statement	Appropriate BASIC instructions
Start	Manual initiation of program by **RUN** and **ENTER**
Establish values of X and Y	**INPUT** X **INPUT** Y
Decrement X	**LET** X = X − 1
Is X = Ø?	**IF** X <> Ø **THEN GOTO** line number
Re-establish X	**LET** X = S (S will have to be set to the initial value of X)
Decrement Y	**LET** Y = Y − 1
Is Y = Ø?	**IF** Y <> Ø **THEN GOTO** line number
Stop	**STOP**

Each instruction in the program obviously requires a line number (it is wise to leave plenty of space between line numbers to allow additional lines to be inserted if necessary), and the correct line numbers must be inserted in the conditional **IF** statement program lines. Also you will have noted that the initial value of X must be saved, because it must be re-established during program execution, and this is achieved by letting S = X immediately after X is put in.

The program is listed below; **PRINT** statements have been included to display suitable captions to assist the user.

```
 5 PRINT "TIME DELAY DEMONSTRATION"
15 PRINT
25 PRINT "INPUT POSITIVE INTEGER X"
35 INPUT X
45 PRINT X
55 LET S = X
65 PRINT "INPUT POSITIVE INTEGER Y"
```

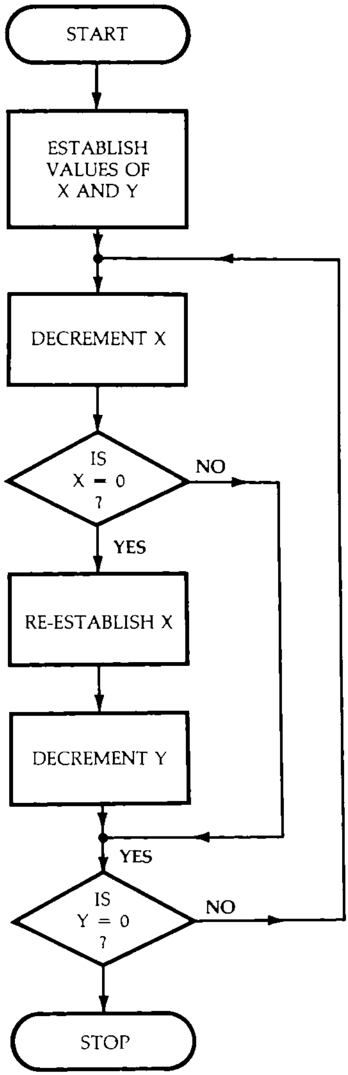


Fig. 5-2. Flowchart for a time delay program.

```
 75 INPUT Y
 85 PRINT Y
 95 LET X = X - 1
105 IF X <> 0 THEN GOTO 135
```

```
115 LET X = S
125 LET Y = Y - 1
135 IF Y <> 0 THEN GOTO 95
145 PRINT "END OF DELAY"
155 STOP
```

Try running this program with X = 10 and Y = 20. You will find that the time required to execute the program is approximately 10 seconds. Try other values of X and Y but do not be tempted to make the numbers too large unless you have plenty of time to wait for the "End of Delay" response.

The above program must have positive integer values for X and Y, and to ensure that noninteger, zero and negative numbers are rejected by the program you can insert the following two additional lines of program:

```
40 IF X < = 0 OR (X - INT X) <> 0 THEN GOTO 25
80 IF Y < = 0 OR (Y - INT Y) <> 0 THEN GOTO 65
```

We have used this time delay program in Program 7, Chapter 8 to generate an approximate one-second delay which is incorporated in a pseudo 24-hour stopwatch. You may find it interesting to try this program for yourself.

What is a subroutine and how is it used?

In many programs you will find that the same set of instructions is required more than once, and in such cases it is sensible to implement this common set of instructions as a subroutine.

A subroutine is a set of program instructions that can be reached *(called)* from more than one place in the main program. (It is normally placed at the beginning or at the end of the main program.) This process is illustrated in Fig. 5-3. To explain the principle let us consider a timer problem which uses a subroutine in implementing a solution.

The problem under consideration is to use the computer to display a positive integer count value which decrements every second until it becomes zero. The displayed count value must indicate the number of seconds remaining in each equal half period, as shown in Fig. 5-4. The display must also indicate which half period is being timed out—see the two graphics symbols in Fig. 5-4. This is imple-

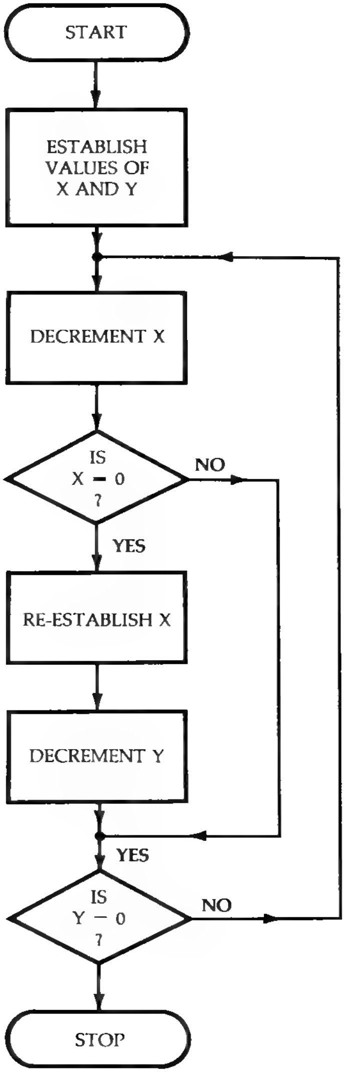


Fig. 5-2. Flowchart for a time delay program.

```
 75 INPUT Y
 85 PRINT Y
 95 LET X = X - 1
105 IF X <> 0 THEN GOTO 135
```

```
115 LET X = S
125 LET Y = Y - 1
135 IF Y <> 0 THEN GOTO 95
145 PRINT "END OF DELAY"
155 STOP
```

Try running this program with X = 10 and Y = 20. You will find that the time required to execute the program is approximately 10 seconds. Try other values of X and Y but do not be tempted to make the numbers too large unless you have plenty of time to wait for the "End of Delay" response.

The above program must have positive integer values for X and Y, and to ensure that noninteger, zero and negative numbers are rejected by the program you can insert the following two additional lines of program:

```
40 IF X < = 0 OR (X - INT X) <> 0 THEN GOTO 25
80 IF Y < = 0 OR (Y - INT Y) <> 0 THEN GOTO 65
```

We have used this time delay program in Program 7, Chapter 8 to generate an approximate one-second delay which is incorporated in a pseudo 24-hour stopwatch. You may find it interesting to try this program for yourself.

What is a subroutine and how is it used?

In many programs you will find that the same set of instructions is required more than once, and in such cases it is sensible to implement this common set of instructions as a subroutine.

A subroutine is a set of program instructions that can be reached *(called)* from more than one place in the main program. (It is normally placed at the beginning or at the end of the main program.) This process is illustrated in Fig. 5-3. To explain the principle let us consider a timer problem which uses a subroutine in implementing a solution.

The problem under consideration is to use the computer to display a positive integer count value which decrements every second until it becomes zero. The displayed count value must indicate the number of seconds remaining in each equal half period, as shown in Fig. 5-4. The display must also indicate which half period is being timed out—see the two graphics symbols in Fig. 5-4. This is imple-

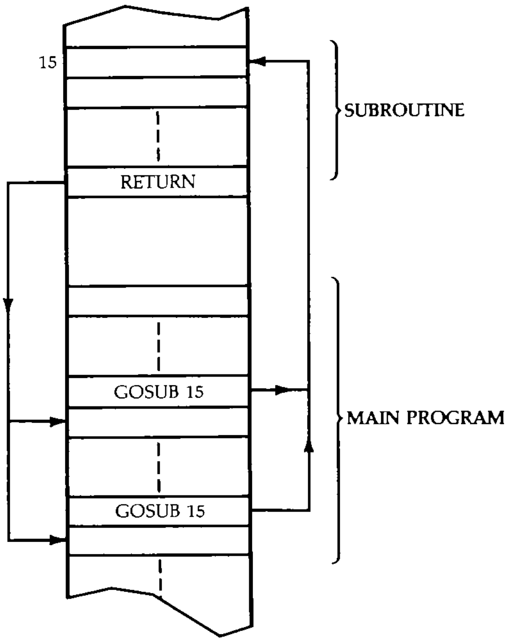


Fig. 5-3. Illustration of subroutine GOSUB and RETURN actions.

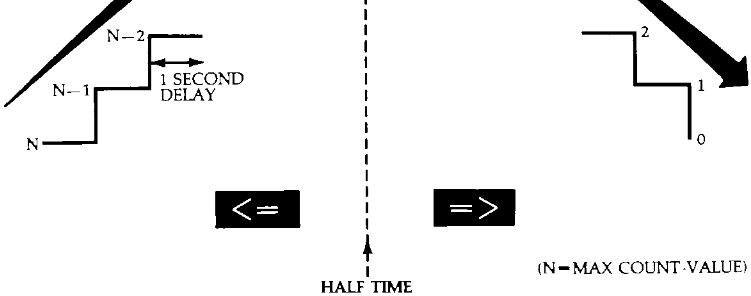


Fig. 5-4. Format of half period time indicator.

mented in the following program: line 30 contains the graphic symbol which indicates the first half period, and line 80 contains the graphic symbol corresponding to the second half period.

In the program the subroutine (lines 170-265) is used to create the required time delay of approximately one second, and you can see that the corresponding common set of instructions is called by using the **GOSUB** statement at two points in the main program, lines 55 and 115.

You may find the program useful as a timer for competitive games or sports which may have predetermined time periods, e.g. chess, ice hockey, Scrabble, etc.

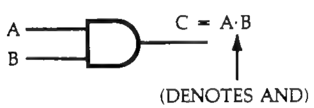
```
  5 PRINT "INPUT MAX INTEGER"
  6 PRINT "COUNT VALUE"
 10 INPUT N
 12 IF N - INT N <> 0 THEN GOTO 135
 20 CLS
 25 FOR J = N TO 0 STEP -1
 30 PRINT "<=";
 55 GOSUB 170
 65 NEXT J
 75 FOR J = N TO 0 STEP -1
 80 PRINT "=>";
115 GOSUB 170
120 NEXT J
125 STOP
135 CLS
145 GOTO 5
170 PRINT J
175 LET X = 14
185 LET S = X
195 LET Y = 1
215 LET X = X - 1
225 IF X <> 0 THEN GOTO 255
235 LET X = S
245 LET Y = Y - 1
255 IF Y <> 0 THEN GOTO 215
260 CLS
265 RETURN
```

CHAPTER 6

Is It Logical?

What are the logic operations AND, OR, and NOT?

The Timex Sinclair 1000/ZX81 uses logic elements (electronic circuits) that give an output dependent on the value of one or more input variables. The input and output values are either logic level 0 (0V) or logic level 1 (+ 5V). The **AND** gate gives an output logic 1 when all the inputs are at logic 1, otherwise the output is logic 0. The symbol and truth table for a two-input **AND** gate are shown in Fig. 6-1.

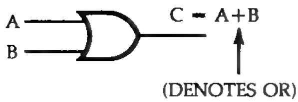


INPUT A B	OUTPUT C
0 0	0
0 1	0
1 0	0
1 1	1

Fig. 6-1. Symbol and truth table for a two-input AND gate.

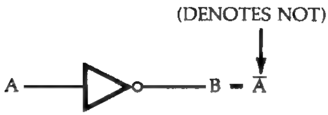
The **OR** gate (inclusive **OR**) gives an output logic 1 when any, or any combinations, of the inputs is/are at logic 1. This implies that the output will only be at logic 0 when all the inputs are at logic 0. The symbol and truth table for a two-input **OR** gate are shown in Fig. 6-2.

The **NOT** gate (inverter or complement gate) gives an output which is the inverted (complemented) logic value of the input. The symbol and truth table are shown in Fig. 6-3.



INPUT A	INPUT B	OUTPUT C
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 6-2. Symbol and truth table for a two-input OR gate.



INPUT A	OUTPUT B
0	1
1	0

Fig. 6-3. Symbol and truth table for the NOT gate.

In the Z80A microprocessor the **AND** and **OR** operations are performed on a bit-by-bit basis between two data bytes. If the two data bytes are A = 10011010 and B = 11100011 the **AND** operation is achieved as

$$\begin{aligned} A &= 10011010 \\ B &= \underline{11100011} \\ \text{Result} = A \bullet B &= \underline{10000010} \end{aligned}$$

and the **OR** operation yields

$$\begin{aligned} A &= 10011010 \\ B &= \underline{11100011} \\ \text{Result} = A + B &= \underline{11111011} \end{aligned}$$

The Z80A microprocessor can perform the **NOT** operation on a data byte, changing 0s to 1s and 1s to 0s. As an example, if A = 01011010 then the one's complement of A, i.e., **NOT** A, is

$$\begin{aligned} A &= 01011010 \\ \text{Result} = \bar{A} &= 10100101 = \text{NOT } A \end{aligned}$$

Note that you can use **AND, OR,** and **NOT** in conditional **IF** statements in your BASIC programs. However, in such cases instead of interpreting the logic operations in terms of 0s and 1s, the Timex Sinclair 1000/ZX81 interprets them as False and True respectively. This can be appreciated by considering the statements

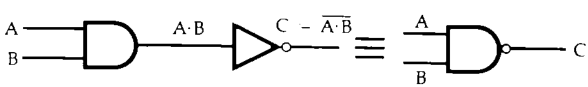
```
14 IF H    = 8 AND H    = 88 THEN GOTO 28
16 STOP
```

where this will be interpreted to mean: if the numeric variable H is in the range 8 to 88 inclusive then go to line 28 of the program, otherwise go to the next line of the program, i.e. to **STOP.** We can represent this in the form of the following truth table.

H > = 8 (H greater than or equal to 8?)	**H < = 88** (H less than or equal to 88?)	**Program control goes to**
True	True	Line 28
True	False	Line 16
False	True	Line 16
False	False	Line 16

What are the logic operations NAND and NOR?

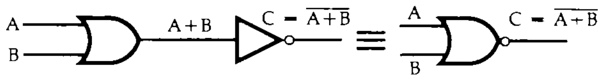
You can implement these logic operations using gate circuits. The **NAND** gate can be obtained by combining an **AND** gate with a **NOT** gate. This is shown in Fig. 6-4 for the case of a two-input **NAND** element. This gate only gives an output logic 0 when all the inputs are at logic 1; for all other input combinations it gives an output logic 1. The symbol and truth table for a two-input **NAND** gate are given in Fig. 6-4.



INPUT A	INPUT B	OUTPUT C
0	0	1
0	1	1
1	0	1
1	1	0

Fig. 6-4. Symbol and truth table for a two-input NAND gate.

The **NOR** gate is obtained by combining an **OR** gate with a **NOT** gate. This is shown in Fig. 6-5 for the case of a two-input **NOR** element. This gate only gives an output logic 1 when all the inputs are at logic 0; for all other input combinations it is given an output logic 0. The symbol and truth table for a two-input **NOR** gate are given in Fig. 6-5.



INPUT		OUTPUT
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Fig. 6-5. Symbol and truth table for a two-input NOR gate.

Let us consider how these logical operations work for two data bytes on a bit-by-bit basis. Using A = 01110101 and B = 11111100 we obtain

$$\begin{array}{lll} & A = & 01110101 \\ & B = & \underline{11111100} \\ \text{Result} = & \overline{A \bullet B} = & \underline{\overline{10001011}} \end{array} \qquad \begin{array}{lll} & A = & 01110101 \\ & B = & \underline{11111100} \\ \text{Result} = & \overline{A + B} = & \underline{\overline{00000010}} \end{array}$$

It is also interesting to note that the **NAND** and **NOR** operations on the numeric variables A and B may be replaced by an equivalent form, obtained using De Morgan's theorems, namely

$$(\text{NAND})\ \overline{A.B} = \overline{A} + \overline{B}$$
$$(\text{NOR})\ \overline{A + B} = \overline{A}.\overline{B}$$

and these are shown in logic diagram form in Fig. 6-6.

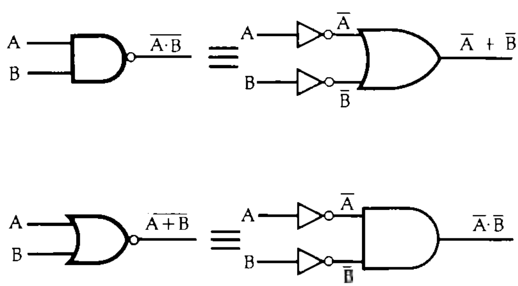


Fig. 6-6. Gate equivalents obtained using DeMorgan's theorems.

What is the logic operation exclusive-OR?

You can implement this operation using a gate circuit. The exclusive-OR gate gives an output logic 1 when any, or any combination, of the inputs, but *excluding* the combination of all inputs, is/are at logic 1. This implies that the output will only be logic 0 when all the inputs are at the same logic level, either logic 1 or logic 0. The symbol and truth table for a two-input exclusive-OR gate are shown in Fig. 6-7.

INPUT A	INPUT B	OUTPUT C
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 6-7. Symbol and truth table for a two-input exclusive-OR gate.

This logic operation can be performed for two data bytes on a bit-by-bit basis. If we consider A = 10011111 and B = 01110001 we obtain

$$\begin{aligned} A &= 10011111 \\ B &= \underline{01110001} \\ \text{Result} = A + B &= \underline{11101110} \end{aligned}$$

The exclusive-OR operation can be performed as a machine-code instruction in the Z80A using an XOR instruction.

What is an IF statement and how is it used?

You may find in a BASIC program that you require a decision to be made regarding the presence of a particular condition. To do this you can use an **IF** statement, which must be written as

IF condition **THEN** statement

You may recall that in the last chapter we met the concept of decision-making, and we saw how to represent this in a flowchart. The decision process directs the program to one of two places. In the BASIC instruction the statement following **THEN** can be used conditionally to direct the program to another line, for example by means of a **GOTO**

statement, othewise if the stated condition is not met the program proceeds to the next line.

The condition of mathematical expressions or numbers can be established using the following mathematical relationships and also shown are the symbols representing them:

equal to:	symbol =
not equal to:	symbol < >
less than:	symbol <
less than or equal to:	symbol < =
greater than:	symbol >
greater than or equal to:	symbol > =

The program listed below provides a method of investigating the effects of using the above relationships.

```
 5 PRINT
15 PRINT "INPUT VALUES FOR A AND B"
25 INPUT A
35 INPUT B
45 CLS
55 IF condition THEN GOTO 85
65 PRINT "CONDITION FALSE"
75 GOTO 5
85 PRINT "CONDITION TRUE"
95 GOTO 5
```

We suggest you enter the program and for your first attempt insert between **IF** and **THEN** the condition A = B in line 55 of the program. On running the program you will note that for entered values of A equal to B the program goes to line 85, while for values of A not equal to B the program goes to line 65.

Now try changing line 55 in the program to test each of the other conditional relationships:

A < > B
A < B
A < = B
A > B
A > = B

It is also possible to use the conditional **IF** statement to determine the condition of strings. The conditional relation-

ships considered above can be used to establish whether two strings are identical (equal) or whether the alphabetical order of one string precedes that of another string. Assuming that the two strings involved with the **IF** statement are A$ and B$, we can summarize the possible conditional tests as follows:

A$ = B$: A$ is identical to B$
A$ < > B$: A$ is not identical to B$
A$ > B$: A$ follows B$ in alphabetical order
A$ < B$: A$ precedes B$ in alphabetical order
A$ > = B$: A$ follows B$ in alphabetical order, or A$ is identical to B$
A$ < = B$: A$ precedes B$ in alphabetical order, or A$ is identical to B$

For example, if we use

```
55 IF A$ < B$ THEN GOTO 85
65 next line of program
```

this is interpreted by the computer to mean: if the string A$ precedes the string B$ in alphabetical order, then go to program line number 85, otherwise go to the next line (line number 65) of the program.

The program listed below provides a method of investigating the above conditional tests on the two strings.

```
 5 PRINT
15 PRINT "INPUT STRINGS FOR A$ AND B$"
25 INPUT A$
35 INPUT B$
45 CLS
55 IF condition THEN GOTO 85
65 PRINT "CONDITION FALSE"
75 GOTO 5
85 PRINT "CONDITION TRUE"
95 GOTO 5
```

It is possible to combine decision-making relationships by using logical operations. The Timex Sinclair 1000/ZX81 BASIC has the logical operations **AND, OR** and **NOT** available for this purpose.

The **AND** operation is used to test that *all* relationships used in the conditional **IF** statement are true. For example,

if we change line 55 in the above program to

55 **IF** A$ = "Z" **AND** B$ = "W" **THEN GOTO** 85

the program goes to line 85 only when the string A$ is equal to the single character Z *and* the string B$ is equal to the single character W, otherwise the next line of the program is used.

The **OR** operation is used to test that at *least one* relationship used in the conditional **IF** statement is true. For example, if we change line 55 in the previous program to

55 **IF** A$ = "Z" **OR** B$ = "W" **THEN GOTO** 85

the program goes to line 85 when *either* the string A$ is equal to the single character Z *or* the string B$ is equal to the single character W, *or* both conditions are true, otherwise the next line of the program is used.

The **NOT** function is used to test that the relationship following it is not true. For example, if we change line 55 in the previous program to

55 **IF NOT** A$ = "Z" **AND** B$ = "W" **THEN GOTO** 85

the program goes to line 85 when the string A$ is *not* equal to the single character Z *and* the string B$ is equal to the single character W, otherwise the next line of the program is used.

CHAPTER 7

Graphics

What is the Timex Sinclair 1000/ZX81 character set?

In Table 7-1 the character set is summarized. You will see that each character is assigned a unique decimal code in the range 0 to 255, although not all codes are used.

In applications where we require the given code of the first character in a string, we use the **CODE** function, while to obtain a character from its code we use the **CHR$** function. If you execute

```
PRINT CODE "P"
```

you will see that the corresponding code 53 is displayed on the screen. If you execute

```
PRINT CHR$ 160
```

you will see that the inverse 4 (white 4 on black background) is displayed on the screen.

You can examine the complete character set conversion from character to code by using

```
 5 INPUT A$
15 PRINT "   "; CODE A$;
25 GOTO 5
```

To examine the conversion from code to character use

```
 5 INPUT A
15 PRINT "   "; CHR$ A;
25 GOTO 5
```

To display the 22 graphics symbols (see codes 0 to 10, and 128 to 138 in Table 7-1) or to display the valid inverse characters (see codes 139 to 191 in Table 7-1) you can use the **GRAPHICS** mode when entering the symbols or inverse characters in the appropriate BASIC program lines. To establish the graphics mode press **GRAPHICS.** The machine will remain in this mode until you press **GRAPHICS** again or until you press **ENTER.** Remember that to obtain the 22 graphics symbols when in the graphics mode you must press the appropriate *shifted* key.

Table 7-1. Character Set Summary

Code	Character	Hex	Code	Character	Hex
0	space	00	33	5	21
1	▘	01	34	6	22
2	▝	02	35	7	23
3	▀	03	36	8	24
4	▖	04	37	9	25
5	▌	05	38	A	26
6	▞	06	39	B	27
7	▛	07	40	C	28
8	▒	08	41	D	29
9	▒	09	42	E	2A
10	▒	0A	43	F	2B
11	"	0B	44	G	2C
12	£	0C	45	H	2D
13	$	0D	46	I	2E
14	:	0E	47	J	2F
15	?	0F	48	K	30
16	(	10	49	L	31
17	)	11	50	M	32
18	>	12	51	N	33
19	<	13	52	O	34
20	=	14	53	P	35
21	+	15	54	Q	36
22	−	16	55	R	37
23	*	17	56	S	38
24	/	18	57	T	39
25	;	19	58	U	3A
26	,	1A	59	V	3B
27	.	1B	60	W	3C
28	0	1C	61	X	3D
29	1	1D	62	Y	3E
30	2	1E	63	Z	3F
31	3	1F	64	**RND**	40
32	4	20	65	**INKEY$**	41

Code	Character	Hex	Code	Character	Hex
66	**PI**	42	157	inverse 1	9D
67 to 111	not used		158	inverse 2	9E
112	cursor up ↑	70	159	inverse 3	9F
113	cursor down ↓	71	160	inverse 4	A0
114	cursor left ←	72	161	inverse 5	A1
115	cursor right →	73	162	inverse 6	A2
116	**GRAPHICS**	74	163	inverse 7	A3
117	**EDIT**	75	164	inverse 8	A4
118	**ENTER**	76	165	inverse 9	A5
119	**DELETE**	77	166	inverse A	A6
120	K/L mode	78	167	inverse B	A7
121	**FUNCTION**	79	168	inverse C	A8
122	not used	7A	169	inverse D	A9
123	not used	7B	170	inverse E	AA
124	not used	7C	171	inverse F	AB
125	not used	7D	172	inverse G	AC
126	number	7E	173	inverse H	AD
127	cursor	7F	174	inverse I	AE
128	█	80	175	inverse J	AF
129	▟	81	176	inverse K	B0
130	▙	82	177	inverse L	B1
131	▄	83	178	inverse M	B2
132	▜	84	179	inverse N	B3
133	▐	85	180	inverse O	B4
134	▚	86	181	inverse P	B5
135	▗	87	182	inverse Q	B6
136	▒	88	183	inverse R	B7
137	▒	89	184	inverse S	B8
138	▒	8A	185	inverse T	B9
139	inverse ”	8B	186	inverse U	BA
140	inverse £	8C	187	inverse V	BB
141	inverse $	8D	188	inverse W	BC
142	inverse :	8E	189	inverse X	BD
143	inverse ?	8F	190	inverse Y	BE
144	inverse (	90	191	inverse Z	BF
145	inverse)	91	192	“”	C0
146	inverse >	92	193	**AT**	C1
147	inverse <	93	194	**TAB**	C2
148	inverse =	94	195	not used	C3
149	inverse +	95	196	**CODE**	C4
150	inverse −	96	197	**VAL**	C5
151	inverse *	97	198	**LEN**	C6
152	inverse /	98	199	**SIN**	C7
153	inverse ;	99	200	**COS**	C8
154	inverse ,	9A	201	**TAN**	C9
155	inverse .	9B	202	**ASN**	CA
156	inverse 0	9C	203	**ACS**	CB

Table 7-1. Character Set Summary (cont.)

Code	Character	Hex	Code	Character	Hex
204	**ATN**	CC	230	**NEW**	E6
205	**LN**	CD	231	**SCROLL**	E7
206	**EXP**	CE	232	**CONT**	E8
207	**INT**	CF	233	**DIM**	E9
208	**SQR**	D0	234	**REM**	EA
209	**SGN**	D1	235	**FOR**	EB
210	**ABS**	D2	236	**GOTO**	EC
211	**PEEK**	D3	237	**GOSUB**	ED
212	**USR**	D4	238	**INPUT**	EE
213	**STR$**	D5	239	**LOAD**	EF
214	**CHR$**	D6	240	**LIST**	F0
215	**NOT**	D7	241	**LET**	F1
216	**	D8	242	**PAUSE**	F2
217	**OR**	D9	243	**NEXT**	F3
218	**AND**	DA	244	**POKE**	F4
219	< =	DB	245	**PRINT**	F5
220	> =	DC	246	**PLOT**	F6
221	<>	DD	247	**RUN**	F7
222	**THEN**	DE	248	**SAVE**	F8
223	**TO**	DF	249	**RAND**	F9
224	**STEP**	E0	250	**IF**	FA
225	**LPRINT**	E1	251	**CLS**	FB
226	**LLIST**	E2	252	**UNPLOT**	FC
227	**STOP**	E3	253	**CLEAR**	FD
228	**SLOW**	E4	254	**RETURN**	FE
229	**FAST**	E5	255	**COPY**	FF

To illustrate this let us consider how to print EG in inverse characters followed by a single normal (noninverse) space, which is followed by the single graphics symbol with the code number 6. To achieve this we follow these steps:

Step	Action
1	press **PRINT**
2	press shifted P
3	press **GRAPHICS**
4	press **EG**
5	press **GRAPHICS**
6	press **SPACE**
7	press **GRAPHICS**
8	press shifted T
9	press **GRAPHICS**
10	press shifted P
11	press **ENTER**

Obviously the character set is useful in creating graphical images on the screen. But it is also worth noting that the computer can operate at two speeds, slow and fast. The slow mode is automatically selected when the microcomputer is switched on and in this mode it is able to compute and display. The fast mode is selected by the **FAST** key (shifted F) and the machine will remain in this mode of operation until the slow mode is reselected by the **SLOW** key (shifted D). In the fast mode the computer gives priority to computations and will only give a display at the end of computations or when it has time to do so.

Where can I print graphic characters?

The visual display is arranged as a 22 × 32 character-grid as shown in Fig. 7-1. The rows are numbered from 0 to 21 starting from the top, and the columns are numbered from 0 to 31 starting from the left side. You can print a character or string of characters at defined locations on the grid by using the keyword **PRINT** followed by the **AT** pseudo-function in the format

PRINT AT row number, column number; "characters"

or alternatively

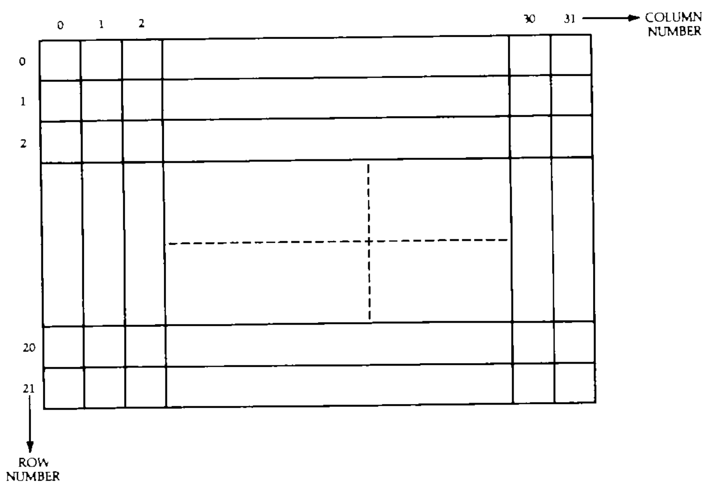


Fig. 7-1. Graphics character-grid.

PRINT AT row number, column number; **CHR$** code number

Fig. 7-2 shows a grid summary of a simple graphics display. This can be displayed using the program

```
 5 PRINT AT 8, 15; "END"
15 PRINT AT 10, 15; CHR$ 23; CHR$ 128; CHR$ 23
```

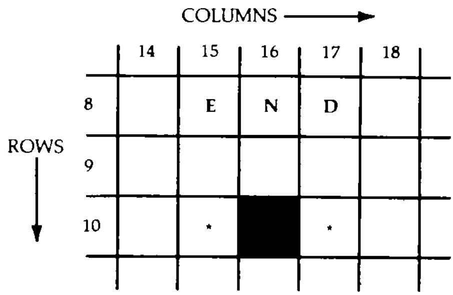


Fig. 7-2. Grid summary of simple graphics display.

To move the **PRINT** position *n* columns along a specified row from the lefthand side of the grid you can use the pseudo-function **TAB.** This has the format **TAB** *n,* where *n* is the number (in the range 0 to 31) specifying the start position of the desired display.

The program listed below illustrates use of the **TAB** pseudo-function to display the character-grid graphics shown in Fig. 7-3.

```
 5 PRINT AT 8, 15; "END"
15 PRINT AT 10, 13; CHR$ 23; TAB 16; CHR$ 128; TAB 19;
   CHR$ 23
```

Two useful statements for graphic applications are **CLS** and **SCROLL.** The **CLS** statement clears all graphics from the screen, and the **SCROLL** statement shifts the entire graphics up one row on the screen grid.

The program below demonstrates use of the **CLS** and **SCROLL** statements. Try the program for yourself.

```
 5 PRINT AT 8, 15; "END"
15 PAUSE 100
25 CLS
```

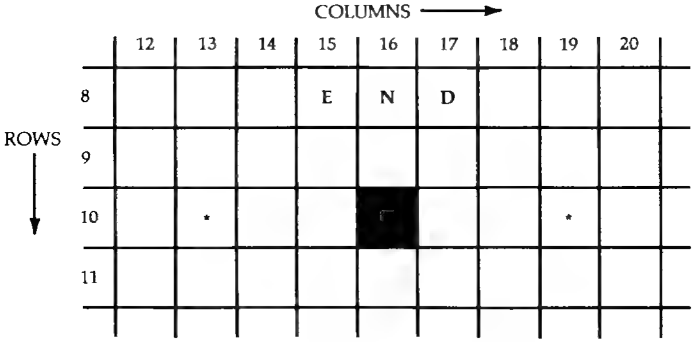


Fig. 7-3. Grid summary for pseudo-function TAB example.

```
35 PRINT AT 10, 13; CHR$ 23; TAB 16; CHR$ 128; TAB 19;
   CHR$ 23
45 PAUSE 100
55 SCROLL
65 PAUSE 100
75 GOTO 55
```

What are screen pixels and how are they used?

For graphics applications the visual display is arranged as a 44 × 64 grid of square picture elements, which are known as pixels. The pixel grid overlays the character grid, such that each square in the character grid contains four pixels. The columns in the x-direction are numbered from 0 to 63 starting from the lefthand side, and the rows in the y-direction are numbered from 0 to 43 starting from the bottom. The pixel-grid arrangement is shown in Fig. 7-4.

The Timex Sinclair 1000/ZX81 can print (black-in) a pixel at a defined location on the pixel grid using the keyword **PLOT** followed by the x and y direction grid numbers, using the format

PLOT x-direction grid number, y-direction grid number

A pixel may be blanked out by using the keyword **UNPLOT** with the format

UNPLOT x-direction grid number, y-direction grid number

To illustrate the use of these keywords consider a simple

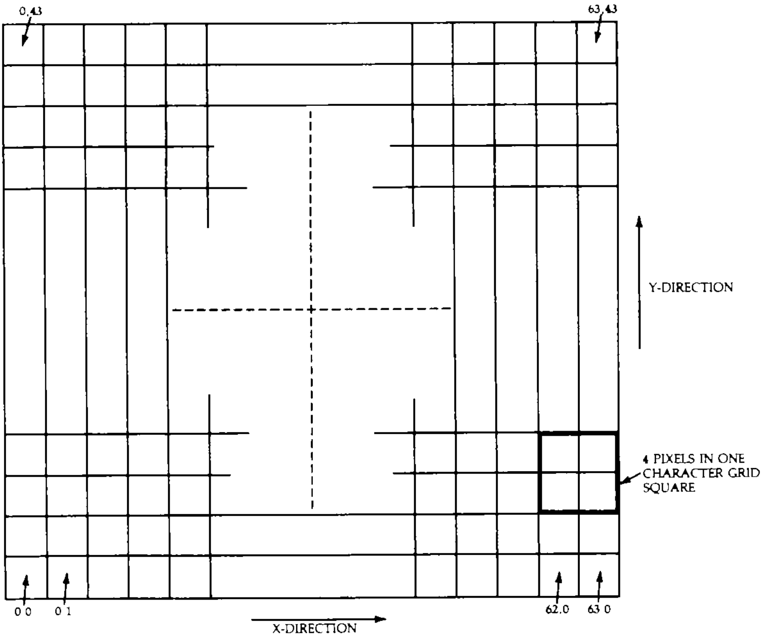


Fig. 7-4. Graphics pixel-grid.

program to plot a 4-pixel by 4-pixel black square in the center of the pixel grid in which the four pixels in the middle of the square have been blanked out. The program listing is

```
25 FOR M = 31 TO 34
35 FOR N = 20 TO 23
45 PLOT M, N
55 NEXT N
65 NEXT M
75 FOR S = 32 TO 33
85 FOR T = 21 TO 22
95 UNPLOT S, T
115 NEXT T
125 NEXT S
```

The ability of the Timex Sinclair 1000/ZX81 to plot and unplot pixels, together with its ability to print characters, provides a powerful graphics facility which is well demonstrated in Program 8, Chapter 8. Run this program and meet Mr. Graphics.

Five machine-code routines, which can be used in graphics applications, are given in Chapter 12, Programs 2 to 6.

How can I create moving graphics?

We have seen how to display a graphic character in a specified character-grid square using **PRINT AT** X, Y; "character", where X and Y represent the row and column numbers respectively. If we keep X constant and increment Y from 0 to 31, then the character will be displayed in every column of row X. For example, the program

```
 5 FOR Y = 0 TO 31
15 PRINT AT 10, Y; "*"
25 NEXT Y
```

displays a line of asterisks in row 10. When you run this program you will note that the graphics character moves from left to right, but leaves an image along its path. The *trailing image* can be removed by changing line 15 to

```
15 PRINT AT 10, Y; " *"
```

that is, one space has been included immediately in front of the asterisk. You may note that the program now displays a moving asterisk starting from column 1 and finishing in column 0 after moving through the columns.

To reverse the direction of movement, say when the asterisk reaches the right side of the screen, the program may be written to detect when Y equals a specified upper limit, and then Y may be decremented to reverse the direction of movement. To avoid a trailing image created by the reversal of direction a space should be included after the asterisk.

If you run the following program you will see that the asterisk bounces back and forth across the screen.

```
15 FOR Y = 0 TO 30
25 PRINT AT 10,Y;"*"
30 IF Y = 30 THEN GOTO 45
35 NEXT Y
45 FOR Y = 30 TO 0 STEP -1
55 PRINT AT 10,Y;"* "
60 IF Y = 0 THEN GOTO 15
65 NEXT Y
```

To achieve vertical movement of the asterisk, the Y (column) parameter is kept constant and the X (row) parameter is varied according to the movement required. Of course, **PRINT AT** statements must be included to display spaces above and below the asterisk to remove the trailing images. Furthermore, in some applications it may be desirable to control movement of the graphics from the keyboard. To investigate vertical movement of the asterisk controlled from the keyboard we suggest that you run the following program. By pressing the U or D key the asterisk will move up or down respectively between the upper and lower limits of the screen.

```
15 LET X = 10
25 LET Y = 15
35 LET K$ = INKEY$
45 IF K$ = "U" THEN LET X = X - 1
50 IF X < 0 THEN LET X = 0
55 IF K$ = "D" THEN LET X = X + 1
60 IF X > 20 THEN LET X = 20
75 PRINT AT X - 1,Y;" "
85 PRINT AT X,Y;"*"
95 PRINT AT X + 1,Y;" "
115 GOTO 35
```

To study the graphics capability further, let us consider how to achieve movement of the displayed asterisk on an approximately circular path on the screen. In this case the X and Y parameters must be evaluated in turn for each required display position. In the following program the coordinates of the character grid square (which lies on the circumference of the circle) are evaluated in lines 45 and 55, and the asterisk is then printed at the required position on the screen. Note how the two **LET** statements in lines 36 and 37 are used to save the "old display position," which is subsequently used in line 58 to wipe out the trailing image.

Run the program and see how the asterisk moves around the screen.

```
 5 LET H = 0
 7 LET U = 0
35 FOR A = 0 TO (2*PI) STEP PI/8
36 LET M = H
37 LET N = U
45 LET H = 15 + INT (8*COS A)
55 LET U = 10 + INT (8*SIN A)
58 PRINT AT N,M;" "
65 PRINT AT U,H;"*"
75 NEXT A
95 GOTO 35
```

A practical application, involving some of the principles discussed in this question, is given in Program 9, in Chapter 8.

CHAPTER 8

Try These Programs

These programs have been included to demonstrate many of the principles described in Chapters 3 to 7. By running and studying the programs you will gain further hands-on operating experience and valuable insight into some of the capabilities of your Timex Sinclair 1000/ZX81. All the programs in this chapter can be run on the 1K ZX81, and we hope that you will find them useful.

1. Decimal to binary/hexadecimal conversion

This program converts positive integer decimal numbers, in the range 0 to 1 048 575, to either binary or hexadecimal equivalents. The answer is displayed as a twenty-digit binary or hexadecimal number.

```
 5 CLS
10 PRINT "INPUT POSITIVE DEC. NO."
25 PRINT
30 INPUT D
32 IF D < 0 THEN GOTO 300
33 CLS
35 IF D > 1048575 THEN GOTO 300
36 PRINT
40 PRINT "INPUT 2 FOR BINARY"
42 PRINT "OR 16 FOR HEXADECIMAL"
45 DIM B$ (20)
50 INPUT B
```

```
52 IF B = 2 THEN GOTO 68
54 IF B = 16 THEN GOTO 68
56 CLS
60 PRINT
65 GOTO 40
68 CLS
69 PRINT
70 FOR I = 1 TO 20
80 LET X = D/B
90 LET R = D-(INT (X)*B)
100 LET B$(I) = CHR$ (28 + R)
110 LET D = INT (X)
120 NEXT I
130 PRINT
140 PRINT "ANSWER IS  ";
160 FOR I = 20 TO 1 STEP-1
170 PRINT B$(I);
180 NEXT I
190 PRINT
195 PRINT
200 PRINT "PRESS G TO GO AGAIN"
210 PRINT "OR S TO STOP"
220 INPUT K$
230 IF K$ = "S" THEN STOP
240 IF K$ = "G" THEN GOTO 5
250 CLS
260 GOTO 200
300 CLS
305 PRINT "OUT OF RANGE"
306 PRINT "TRY AGAIN"
310 GOTO 10
```

2. Binary/hexadecimal to decimal conversion

This program converts either a positive integer binary or a positive integer hexadecimal number to the corresponding decimal equivalent.

```
10 PRINT "ENTER RADIX: 2 OR 16"
20 PRINT
30 INPUT RADIX
40 IF RADIX = 2 THEN GOTO 55
```

```
42 IF RADIX = 16 THEN GOTO 80
44 GOTO 230
55 PRINT "INPUT YOUR BIN. NO."
60 INPUT N$
70 GOTO 94
80 PRINT "INPUT YOUR HEX. NO."
90 INPUT N$
94 LET P = 1
96 LET R = 0
100 LET L = LEN N$
110 FOR K = L TO 1 STEP-1
120 LET A = (CODE (N$ (K))-28)*P
130 LET P = P*RADIX
140 LET R = R + A
150 NEXT K
160 PRINT
162 PRINT "ANSWER =";R
170 PRINT
180 PRINT "PRESS G TO GO AGAIN"
184 PRINT "OR PRESS S TO STOP"
186 PRINT
190 INPUT K$
195 IF K$ = "S" THEN STOP
198 IF K$ <> "G" THEN GOTO 180
230 CLS
240 GOTO 10
```

3. Total cost

This program calculates the total cost of N items, much the same as a cash register in a supermarket.

```
3 PRINT "ENTER NO. OF ARRAY"
4 PRINT "ELEMENTS"
6 INPUT N
8 IF N < 1 THEN GOTO 120
9 PRINT N
14 DIM T(1)
16 DIM C(N)
18 PRINT "ENTER COST OF ITEMS"
20 FOR I = 1 TO N
30 INPUT C(I)
```

```
35 IF C(I) < 0 THEN GOTO 120
50 PRINT C(I); "   ";
70 LET T(1) = T(1) + C(I)
80 NEXT I
85 PRINT
95 PRINT "TOTAL COST IS $"; T(1)
110 STOP
120 CLS
140 GOTO 3
```

4. Total personal and item cost

This program uses a 20-element, two-dimensional array to store the cost of five separate items for four people. It calculates the total expenditure per person and the total expenditure by the four people on each item.

```
20 DIM R(5)
30 DIM T(4)
40 DIM C(4, 5)
50 FOR I = 1 TO 4
60 FOR J = 1 TO 5
66 PRINT "INPUT C(";I;",";J;")"
67 INPUT C(I,J)
68 IF C(I,J) < 0 THEN GOTO 66
71 LET T(I) = T(I) + C(I,J)
75 CLS
80 NEXT J
90 NEXT I
100 FOR J = 1 TO 5
120 FOR I = 1 TO 4
140 LET R(J) = R(J) + C(I,J)
150 NEXT I
180 NEXT J
200 PRINT "TOTAL COSTS"
210 PRINT
220 PRINT "ROWS"
230 FOR I = 1 TO 4
250 PRINT "   ", T(I),
260 NEXT I
265 PRINT
270 PRINT "COLUMNS"
```

```
275 PRINT
280 FOR J = 1 TO 5
300 PRINT "  "; R(J);
310 NEXT J
```

5. Care for a drink?

The program uses string variables to create a limited dictionary of words which are concatenated to display messages useful for a drink dispensing machine or a menu.

```
 10 LET A$ = "AND"
 20 LET B$ = "BLACK"
 30 LET C$ = "COFFEE"
 40 LET W$ = "WITH"
 50 LET S$ = "SUGAR"
 60 LET N$ = "NO"
 70 LET M$ = "MILK"
 80 LET G$ = "  "
 90 CLS
100 PRINT "WE HAVE"
120 PRINT "1 ";
130 LET K$ = B$ + G$ + C$ + G$ + W$ +G$ + S$
140 PRINT K$
150 PRINT "2 ";
160 LET L$ = B$ + G$ + C$ + G$ + N$ + G$ + S$
170 PRINT L$
180 PRINT "3 ";
190 LET H$ = C$ + G$ + W$ + G$ + M$ + G$ + A$ +
    G$ + S$
200 PRINT H$
240 PRINT
250 PRINT "ENTER 1, 2 OR 3 TO SELECT"
260 INPUT X
270 IF X>3 THEN GOTO 90
280 IF X<1 THEN GOTO 90
290 CLS
300 PRINT "YOUR SELECTION IS "
310 IF X = 1 THEN PRINT K$
320 IF X = 2 THEN PRINT L$
330 IF X = 3 THEN PRINT H$
```

6. Die

Here is a program which provides you with an electronic die, or in other words one of a pair of dice.

```
 15 PRINT "ENTER N FOR NEXT NO."
 25 INPUT A$
 35 CLS
 45 IF A$ <> "N" THEN GOTO 15
 55 RAND
 65 LET A = 1E8*RND
 75 LET B = INT A
 85 LET Y = INT (7*(A - B))
 95 IF Y = 0 THEN GOTO 55
115 PRINT "THE DIE NO. IS "; Y
125 PRINT
135 GOTO 15
```

7. Rolf's watch

A pseudo 24-hour stop watch, which uses an approximate one-second time period to update it, is implemented by this program. The watch has to be initialized, and this can be any time in hours, minutes and seconds within the normal 24-hour period. The watch is then started and subsequently halted by pressing the S and H keys respectively. When halted it displays the time with respect to its initialized value. To use it as a straightforward stop watch you simply initialize the watch to zero, and then when subsequently halted it displays the elapsed time.

```
 5 PRINT "INPUT HRS"
15 INPUT H
25 PRINT "INPUT MINS"
35 INPUT M
45 PRINT "INPUT SECS"
55 INPUT SEC
60 IF NOT (H < = 23 AND M < = 59 AND SEC < = 59)
   THEN GOTO 285
64 PRINT "KEY S/H TO START/HALT WATCH"
66 IF INKEY$ <> "S" THEN GOTO 66
75 CLS
85 LET N = 0
```

```
 95 PRINT "HRS  MINS  SECS"
105 LET X = 14
115 LET S = X
125 IF N = 2 THEN GOTO 75
135 LET N = N + 1
145 LET Y = 1
155 LET X = X - 1
165 IF X <> 0 THEN GOTO 195
175 LET X = SEC
185 LET Y = Y - 1
195 IF Y <> 0 THEN GOTO 155
205 IF SEC = 60 THEN LET M = M + 1
215 IF SEC = 60 THEN LET SEC = 0
225 IF M = 60 THEN LET H = H + 1
235 IF M = 60 THEN LET M = 0
245 IF H = 24 THEN LET H = 0
265 LET SEC = SEC + 1
270 IF INKEY$ = "H" THEN GOTO 355
275 GOTO 105
285 CLS
295 GOTO 5
355 PRINT H;"   ";M;"   ";SEC
365 STOP
```

8. Mr. Graphics

The specification for Mr. Graphics is shown in Fig. 8-1. This includes printed graphic characters and plotted pixels. Furthermore he winks at you!

```
15 FOR J = 26 TO 39
25 FOR K = 16 TO 33
35 PLOT J,K
45 NEXT K
55 NEXT J
65 PRINT AT 10,16;CHR$ 186
67 PRINT AT 3,9;"HELLO ZX81 USERS"
68 PRINT AT 7,12;CHR$ 144
69 PRINT AT 7,20;CHR$ 145
75 PRINT AT 7,14;CHR$ 149
85 FOR I = 15 TO 17
95 PRINT AT 12,I;CHR$ 131
```

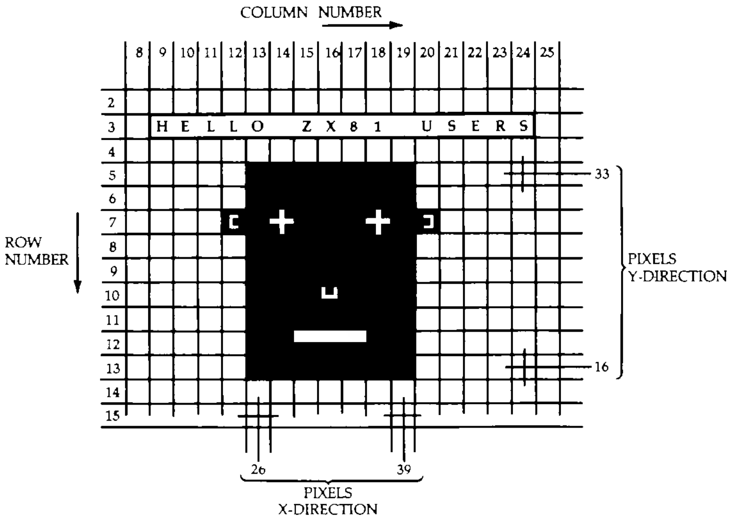


Fig. 8-1. Specifications of Mr. Graphics.

```
115 NEXT I
125 PRINT AT 7,18;CHR$ 149
135 GOSUB 155
145 PRINT AT 7,18;CHR$ 128
148 GOSUB 155
151 GOTO 125
155 LET X = 7
185 LET X = X - 1
195 IF X <> 0 THEN GOTO 185
245 RETURN
```

You may note that the winking effect is achieved by using a time delay subroutine. The alternative approach using **PAUSE** statements will not necessarily achieve a satisfactory flicker-free display. You may investigate this by replacing lines 125 to 245 in the above program by

```
125 GOSUB 155
135 GOTO 125
155 PRINT AT 7,18;CHR$ 149
165 PAUSE 30
175 PRINT AT 7,18;CHR$ 128
185 PAUSE 30
195 RETURN
```

9. Hit the target

This is a program involving moving graphics and it is a game for two players. One player is required to enter the row and column numbers corresponding to a screen grid square (see Fig. 7-1) which is to be the invisible target. Of course, this must be done without the opponent seeing the entered parameters.

After entering the target coordinates an asterisk is displayed in the center of the screen, and the opponent is then required to move the asterisk to try to hit the target. The asterisk is moved up or down by pressing the U or D key respectively, or it is moved left or right by pressing the L or R key respectively. When the asterisk is moved into the square corresponding to the target coordinates a HIT TARGET message is displayed. The game can now be repeated by pressing the G key.

The person taking the least time to score a hit is deemed to be the winner.

The program listing for the game is given below.

```
3 CLS
5 PRINT "INPUT ROW NO."
6 INPUT R
7 PRINT R
8 PRINT "INPUT COLUMN NO."
9 INPUT C
10 PRINT C
15 LET X = 10
25 LET Y = 15
35 LET K$ = INKEY$
45 IF K$ = "U" THEN LET X = X - 1
50 IF X < 0 THEN LET X = 0
55 IF K$ = "D" THEN LET X = X + 1
60 IF X > 21 THEN LET X = 21
65 IF K$ = "L" THEN LET Y = Y - 1
70 IF Y < 0 THEN LET Y = 0
75 IF K$ = "R" THEN LET Y = Y + 1
80 IF Y > 31 THEN LET Y = 31
82 CLS
95 PRINT AT X,Y;"*"
96 IF X = R AND Y = C THEN GOTO 185
```

```
120 GOSUB 145
125 GOTO 35
145 LET J = 5
160 IF J = 0 THEN RETURN
165 LET J = J - 1
175 GOTO 160
185 CLS
190 PRINT AT 10,10;"HIT TARGET"
200 PRINT AT 12,5;"PRESS KEY G TO GO AGAIN"
225 IF INKEY$ = "G" THEN GOTO 3
235 GOTO 225
```

And finally

To gain further insight and experience in writing BASIC programs we suggest that you try to modify and improve the programs in this chapter.

CHAPTER 9

Some Black Boxes

What are flip-flops, flags, registers, and counters?

Flip-flops are important elements in microcomputers and can be used as memory cells to store binary bits (logic 0 or logic 1).

The simple Set-Reset (SR) flip-flop can be implemented using **NAND** or **NOR** gates. The logic symbol and basic **NAND** gate implementation are shown in Fig. 9-1. Note that when the Set-input is changed from logic 1 to logic 0, the Q-output goes to logic 1 and correspondingly $\overline{Q}$ goes to 0. The outputs stay at these logic levels even after the Set-input returns to logic 1, and will only subsequently give a logic 0 at the Q-output and a logic 1 at the $\overline{Q}$-output after the application of a logic 0 to the Reset-input. This type of flip-flop has ambiguous output states when the S and R inputs are set at 0 and, by design, this state is avoided in practice.

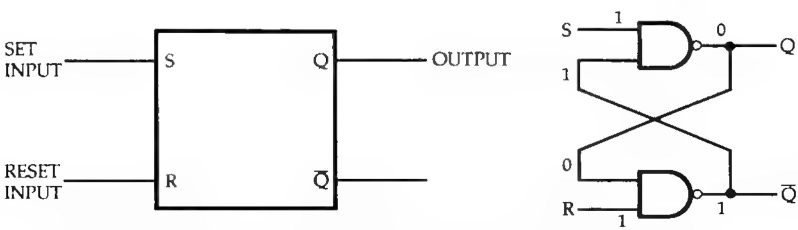


Fig. 9-1. Symbol and NAND gate implementation of set-reset flip-flop.

This flip-flop is often used in applications requiring only the availability of the Q-output and in such cases the flip-flop is referred to as a *status flag,* and the Q-output is called the *flag output.*

Flags are used for indicating the occurrence of events taking place within a microprocessor. For example, if the execution of an arithmetic operation results in the answer being zero, a flag is set in the Z80A microprocessor. If the result is not zero the flag will be reset. The flag in this case is referred to as the Z-flag and is one of the status flags contained in a group within the microprocessor called the *Flag Register.*

The D-type flip-flop is similar to the Set-Reset flip-flop but it has one data input (D), a clock input (C) and two outputs Q and $\overline{Q}$. The binary value (data) to be stored is applied to the D-input and on receipt of the positive-going edge of a clock pulse the binary value on the D-input is stored in the flip-flop and appears at the Q-output. The inverted (negated) value of the D-input appears at the $\overline{Q}$-output. The symbol for this type of flip-flop is shown in Fig. 9-2.

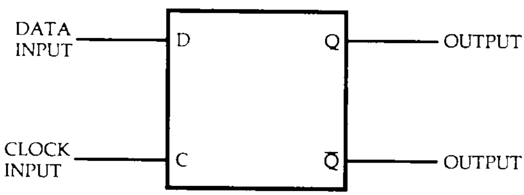
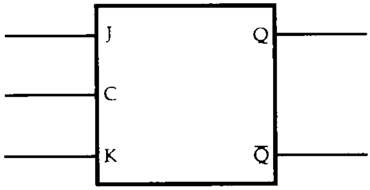


Fig. 9-2. Symbol for D-type flip-flop.

The clocked J-K flip-flop is similar to the Set-Reset flip-flop, because it has two inputs, J and K, and these control the state of the flip-flop in conjunction with an applied clock pulse. The input condition J = K = 1 causes the Q-output to change state (toggle) on receipt of each negative-going edge of the clock pulse. The symbol and function table for a J-K flip-flop are given in Fig. 9-3.

We have seen that a flip-flop can store one bit of information. When flip-flops are organized to store a binary word the arrangement is referred to as a register. If the data is set into and read out of all flip-flops simultaneously, the



INPUTS		OUTPUT
J	K	Q_n+1
0	0	Q_n
1	0	1
0	1	0
1	1	$\bar{Q}_n$

Where Q_n = state of flip-flop before clock pulse
Q_n+1 = state of flip-flop after clock pulse

Fig. 9-3. Symbol and function table for a clocked J-K flip-flop.

register is a parallel-in, parallel-out register, whereas if the data is entered one bit at a time and read out of all flip-flops simultaneously, the register is a serial-in, parallel-out register.

An 8-bit parallel-in, parallel-out register, constructed using D-type flip-flops, is shown in Fig. 9-4. When the write line to the register is pulsed positively the 8-bit binary input word is stored in the flip-flops. A two-input AND gate is included; one input to each gate is the Q-output from the corresponding D-type flip-flop, the other input to the gate is connected to the read line. Consequently the true output word will only appear on the output data lines when a read signal is applied.

When it is desired to manipulate data in serial form a shift register is used. Fig. 9-5 illustrates an 8-bit shift register implemented using eight J-K flip-flops. The first bit of the data is transferred into the leftmost flip-flop on receipt of the first negative-going edge of the clock pulse. On each successive negative-going edge of the clock pulse each data-bit is shifted right into the next flip-flop, and a new data-bit enters the leftmost flip-flop. It requires eight clock pulses to load an 8-bit serial word into this register. If the output of all eight flip-flops is read after eight clock pulses have been applied, the serial word is available in a parallel form. This is a method of serial to parallel conversion.

It is worth noting that the Z80A microprocessor in your Timex Sinclair 1000/ZX81 contains several 8-bit and 16-bit special-purpose and general-purpose registers, and you will meet these in Chapter 10.

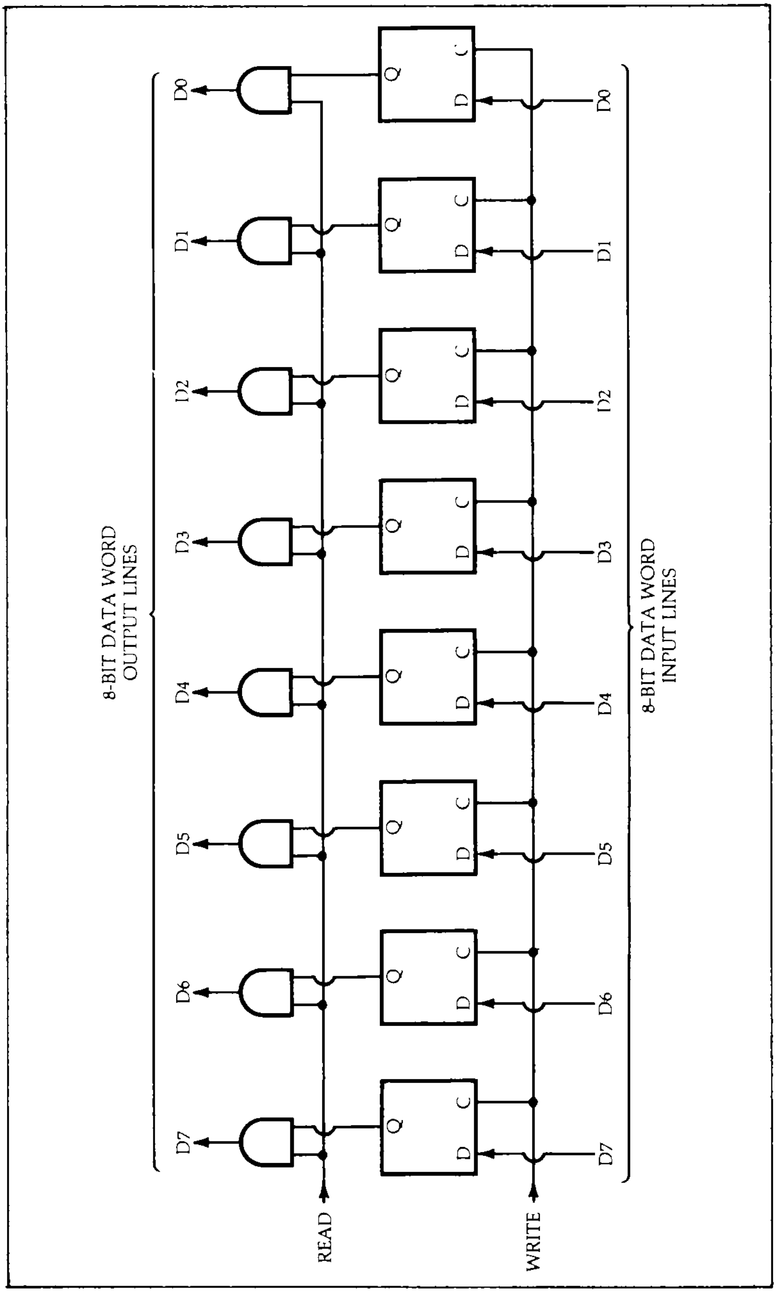


Fig. 9-4. An 8-bit parallel register.

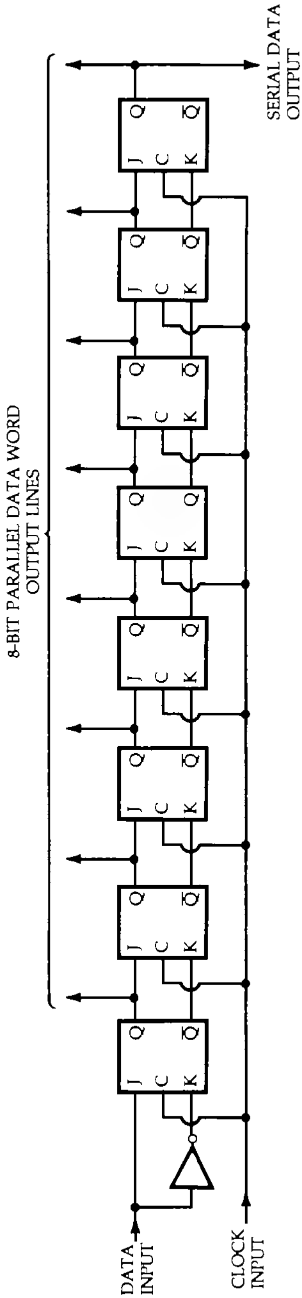


Fig. 9-5. An 8-bit shift register constructed using J-K flip-flops.

Flip-flops can be connected together to form different types of counters. The J-K flip-flop is especially suited for this application because it has a toggle capability when J and K are set to 1. Fig. 9-6 shows how four J-K flip-flops may be connected to form a 0000 to 1111 binary ripple-up counter. We can consider the output of each flip-flop to have a weighted binary value. The first (lefthand) flip-flop has a weight of 1, the second has a weight of 2, the third a weight of 4 and the righthand flip-flop has a weight of 8. This counter continues to count on successive input pulses and on the next clock pulse after reaching 1111 it *rolls over* to 0000.

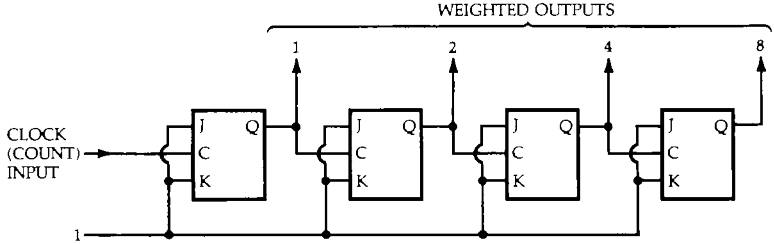


Fig. 9-6. Four-stage binary ripple-up counter.

The addition of extra flip-flops increases the maximum possible count value. A binary ripple-up counter constructed using N flip-flops will be able to count $2^N - 1$ pulses before rolling over.

The Z80A contains an accessible 16-bit Program Counter which contains the address of the next instruction to be fetched from memory. The counter may be preloaded and its content is changed during program execution.

An appreciation of flip-flops, flags, registers and counters helps in understanding the Z80A microprocessor, machine-code programming, and interfacing.

What is random access memory and how is it used?

A random access memory (RAM) is used by the Timex Sinclair 1000/ZX81 to store your program, data and system vari-

ables. It stores this information in the form of binary words (see Chapter 3), with each binary word held in an addressable memory location.

RAM can be thought of as an array of single-bit memory cells, as shown in Fig. 9-7, arranged in groups of eight bits (bytes) forming addressable memory locations. The chip containing the memory-cell array also includes an address decoder, and some control and interface logic.

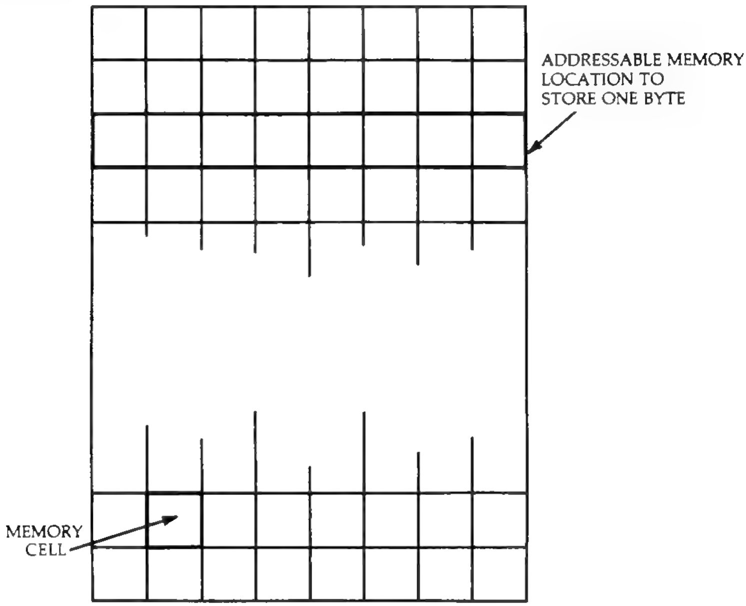


Fig. 9-7. Memory cell array.

Fig. 9-8 shows the schematic of a single-chip integrated circuit 128 × 8-bit RAM. Each of the 128 storage locations has its own unique address and selection is achieved by applying a binary code to address lines A0 to A6. Address codes from 0000000 to 1111111 are possible thus giving 2^7 (128_{10}) combinations. The control of whether data is read from or written to a selected memory location is determined by the logic level on the read/write control line. A logic 1 will initiate a read operation and logic 0 a write operation.

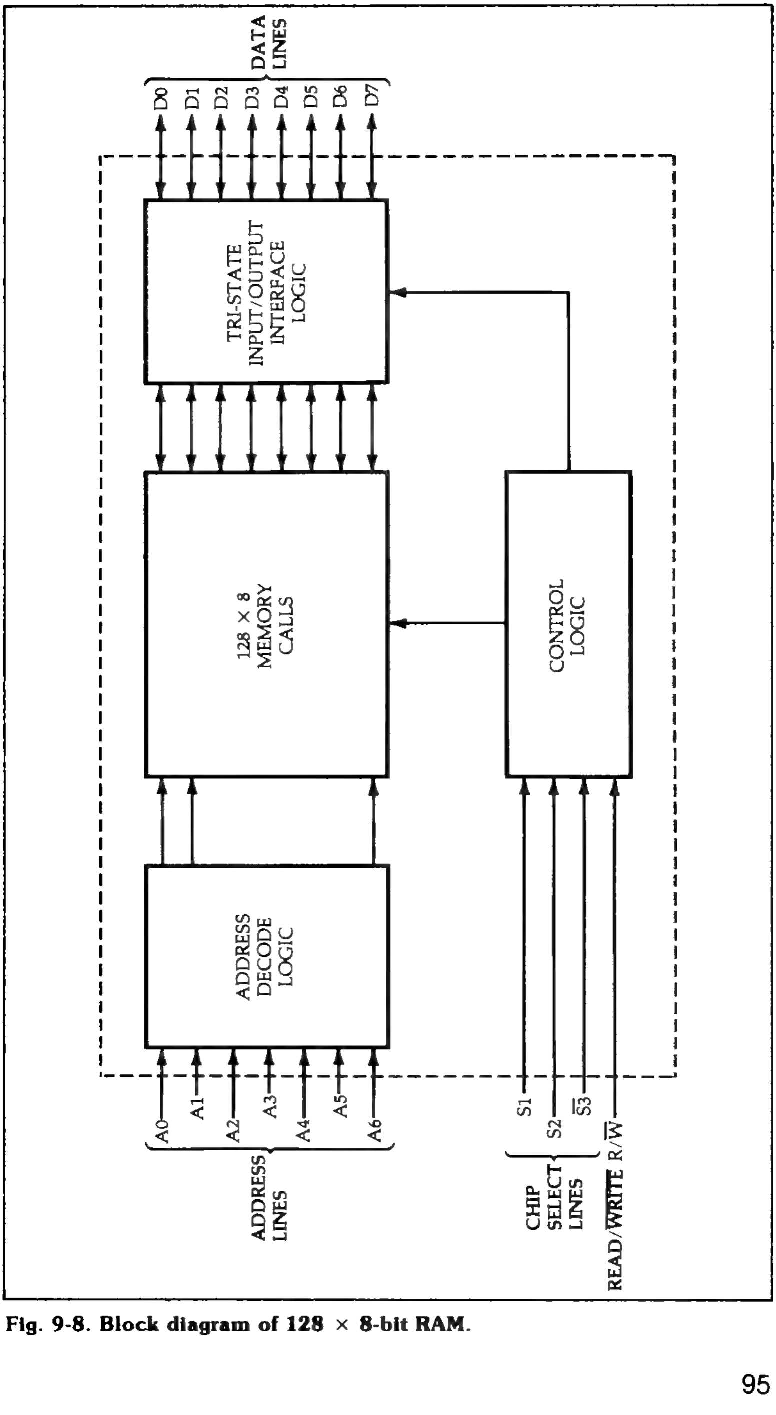


Fig. 9-8. Block diagram of 128 × 8-bit RAM.

To enable several RAM chips to be connected to a common bidirectional data bus, further address lines, called *chip select* lines, are used. The address lines A0 to A6 from the main bus go to a number of identical chips, but only if the binary code on the chip select line is correct will the chip be selected. When a logic 1 is applied to S1 and S2 and a logic 0 is applied to S3 the RAM will be addressed. When not selected the memory output interface goes into the high impedance state which effectively isolates the device from the data bus.

Your machine uses either two RAMS, type 2114, or one RAM, type 4118. The 2114 is a 4096-bit static RAM that is nibble-organized into 1024 words of 4 bits, so that when two 2114 devices are used a 1024 byte-organized memory is obtained. The 4118 is an 8192-bit static RAM, byte-organized into 1024 words of 8 bits.

The pin assignment of the 2114 static RAM chip used in the ZX81 is given in Fig. 9-9. There are four bidirectional tristate input/output data bus connections, I/01 to I/04. The device has ten address bus connections A0 to A9, a single active-low chip select input $\overline{\text{RAMCS}}$, and a single write/read, $\overline{\text{WR}}$, control bus line.

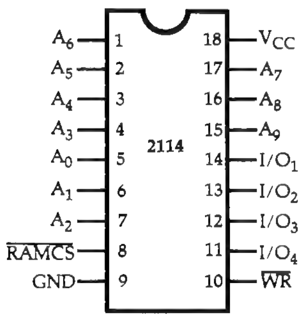


Fig. 9-9. Pin assignment for 2114 RAM.

The byte-organized, 1024-word memory arrangement implemented using two 2114 devices is shown in Fig. 9-10. This is the configuration used as illustrated in Fig. 9-11. Note that the four data lines of IC4A are connected to data

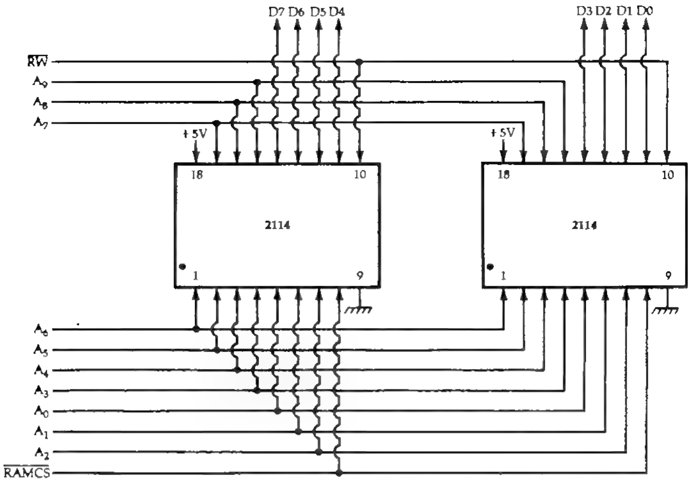


Fig. 9-10. Cascade connection to form 2114 byte memory.

lines D2, D3, D4, and D7 of the data bus, and the four data lines of IC4B are connected to data lines D0, D1, D5, and D6. The address lines A0 to A9 of both devices are connected to the corresponding address bus lines. The $\overline{\text{RAMCS}}$ chip select pin on each device is connected to pin 12 of the special Sinclair Logic Device, IC1, and the write/read, $\overline{\text{WR}}$, pin on each RAM is connected to the $\overline{\text{WR}}$ pins on IC1 and the Z80A microprocessor.

The pin assignment of the 4118 static RAM is shown in Fig. 9-12. There are eight bidirectional and tristate input/output data bus connections, DQ0 to DQ7, and ten address bus connections A0 to A9. The chip select input, $\overline{\text{RAMCS}}$, and the output enable, $\overline{\text{OE}}$, may be connected together, as shown in Fig. 9-11. The write/read, $\overline{\text{WR}}$, input controls the data transfer to and from this RAM.

Note that both types of RAM described above are static, that is, they only require a dc supply to maintain the stored content. The main disadvantage with RAM devices is that they are *volatile* which means that the data will only remain stored for as long as the power supply is present. Both types of RAM have a typical access time of 250 nanoseconds (250 $\times 10^{-9}$ seconds).

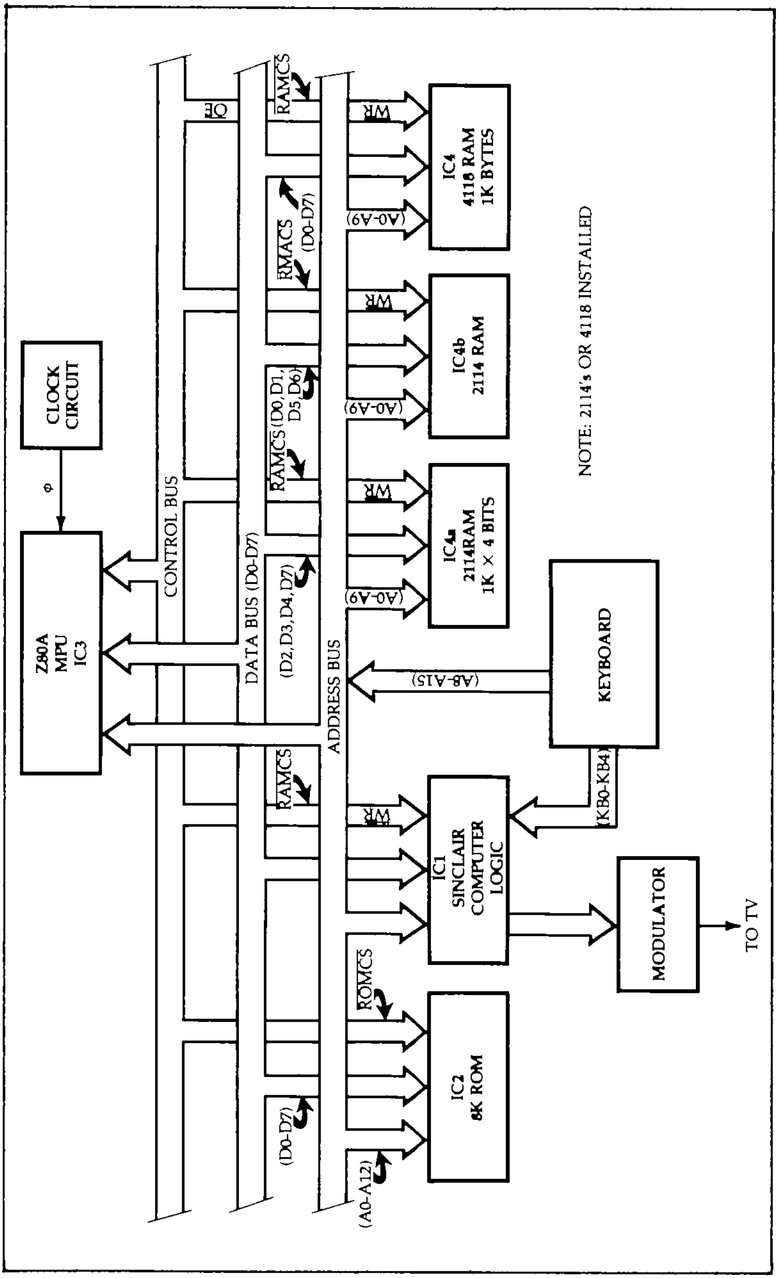


Fig. 9-11. Diagram for Timex Sinclair 1000/ZX81 hardware configuration.

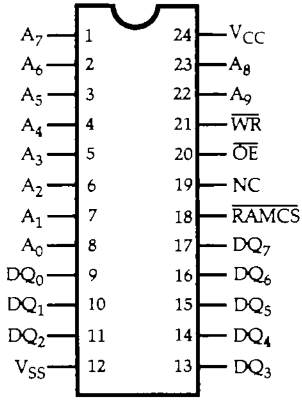


Fig. 9-12. Pin assignment for 4118 RAM.

The two 2114 RAMS *or* the 4118 RAM in your computer have been address decoded to have addresses in the range 16384 to 17407, and you can, of course, as explained in Chapter 3, examine the content of an address in this range using the **PEEK** function or write into a specified address using the **POKE** statement.

The amount of RAM accessible to the BASIC Interpreter is defined by the address held in the system variable known as RAMTOP. It is possible to change the value of RAMTOP, which is useful when machine-code programs are to be implemented. See Chapter 11 for details of how to define RAMTOP and write machine-code programs.

What is read only memory and how is it used?

A Read Only Memory (ROM) is used in the Timex Sinclair 1000/ZX81 to store the 8K (8 × 1024 bytes of memory) Monitor Program. This ROM stores information in the form of binary words (see Chapter 3) with each binary word held in addressable memory location.

The ROM can be considered to be an array of single-bit memory cells arranged in groups of eight bits (bytes) forming addressable memory locations. Each memory cell is programmed during the manufacturing process in accordance with the customer's requirements to store permanently logic 0 or logic 1.

Once programmed the contents of the ROM cannot be changed. The ROM does not need to have power continually applied or to be refreshed, hence the ROM is said to be *non-volatile.*

ROMs are addressed in the same way as RAMs, but the tristate data output lines can only feed data to the data bus. Consequently a single-chip 128 × 8-bit ROM will have the same schematic format as the equivalent sized RAM (Fig. 9-8) except that the data bus lines D0 to D7 will only accept (read) data from the device. Therefore a read/write input signal is not required.

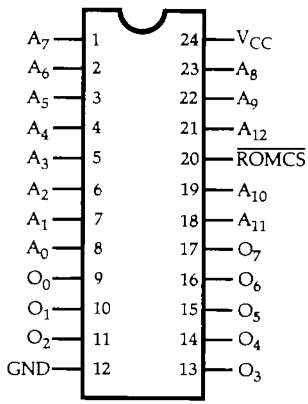


Fig. 9-13. Pin assignment for 2364 ROM.

Your machine uses a ROM, type 2364, which has the pin assignment shown in Fig. 9-13. These are eight tristate output data bus connections O0 to O7 and 13 address connections A0 to A12. It has a single chip select input $\overline{\text{ROMCS}}$ which is connected to pin 13 of the special Sinclair logic device IC1. It has a nominal access time of 450 nanoseconds (450×10^{-9} seconds).

An important point to note is that some of the BASIC command routines stored in the 8K ROM may be usefully incorporated in your machine-code programs. So in using them you save RAM memory space and reduce your machine-code program writing time. This feature is described in Chapter 11.

Why does the Timex Sinclair 1000/ZX81 contain the Sinclair Computer Logic Chip?

The microcomputer contains RAM and ROM, and has facilities for the user to input information from the keyboard and cassette, and to output information to the cassette, printer and TV monitor. To ensure that the system operates correctly under user and program control extensive timing and control circuitry is necessary. It is evident that the Sinclair Computer Logic Chip has been designed to implement the necesary timing and control logic.

Although full details of this chip are not provided with the microcomputer and are not readily available in published literature, it is apparent that the device is used for:

1. chip select and address decode for RAM and ROM;
2. timing and control of the Z80A microprocessor;
3. sensing keyboard operation;
4. control and transfer of input and output of data from and to peripherals (cassette, printer, TV monitor and keyboard).

What is meant by memory being mapped into the system?

When the Z80A microprocessor is addressing memory it will be either accessing a RAM location, when the chip select signal $\overline{\text{RAMCS}}$ is 0v, or a ROM location, when the chip select signal $\overline{\text{ROMCS}}$ is 0v. A location within the selected memory is accessed using the appropriate memory address which is placed on the address bus. Clearly then, each memory location, whether in RAM or in ROM, has a unique address defined by the chip select and address bus lines.

The address connections (shown in Fig. 9-11) and the address decoding logic within the Sinclair Computer logic chip, enable memory access to any one of the 8K locations in the ROM or any of the 1K locations in the RAM (or 16K if you have the 16 Kbyte RAM pack). The corresponding ROM and RAM address ranges are:

ROM	0000 to 8191	
RAM	16384 to 17407	(1K RAM)
	16384 to 32767	(16K RAM)

The specified memory ranges are often shown diagrammatically in the form known as a memory map, which illustrates the specified memory ranges for each system device or peripheral. Fig. 9-14 shows the ROM and RAM memory mapped addresses.

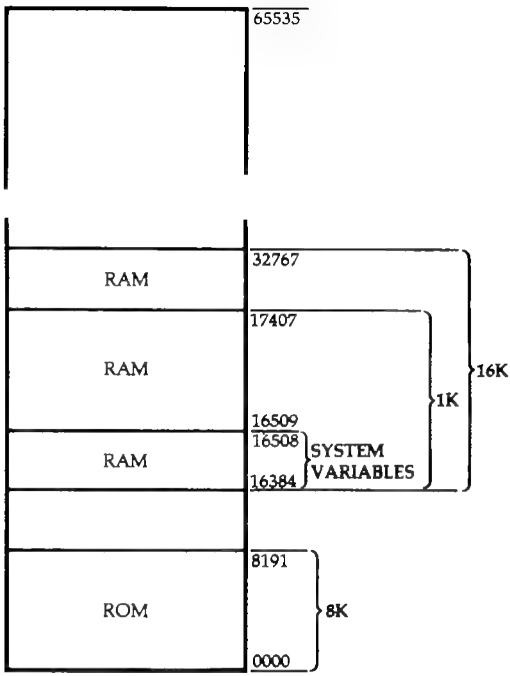


Fig. 9-14. ROM and RAM memory mapped addresses.

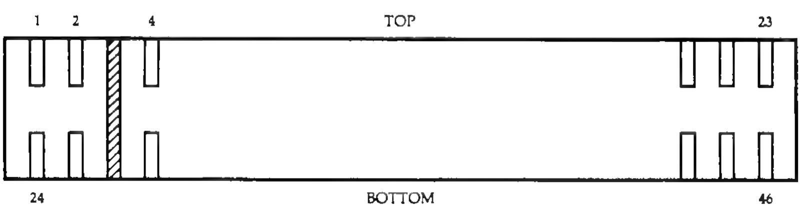


Fig. 9-15. Pin number assignments for edge connector.

Table 9-1. Connections for 46-Way Edge Connector

Pin Number	Signal	Pin Number	Signal
1	D7	24	5V
2	$\overline{\text{RAMCS}}$	25	9V
3	SLOT	26	SLOT
4	D0	27	0V
5	D1	28	0V
6	D2	29	$\overline{\phi}$
7	D6	30	A0
8	D5	31	A1
9	D3	32	A2
10	D4	33	A3
11	$\overline{\text{INT}}$	34	A15
12	$\overline{\text{NAY}}$	35	A14
13	$\overline{\text{HALT}}$	36	A13
14	$\overline{\text{MREQ}}$	37	A12
15	$\overline{\text{IORQ}}$	38	A11
16	$\overline{\text{RD}}$	39	A10
17	$\overline{\text{WR}}$	40	A9
18	$\overline{\text{BUSAK}}$	41	A8
19	$\overline{\text{WAIT}}$	42	A7
20	$\overline{\text{BUSRQ}}$	43	A6
21	$\overline{\text{RESET}}$	44	A5
22	$\overline{\text{MI}}$	45	A4
23	$\overline{\text{REFSH}}$	46	$\overline{\text{ROMCS}}$

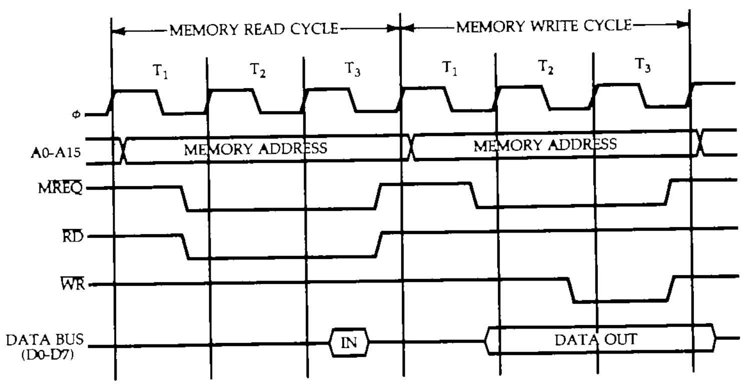


Fig. 9-16. MPU read or write cycles.

How can I interface external hardware?

The address, data and control bus connections are pro-

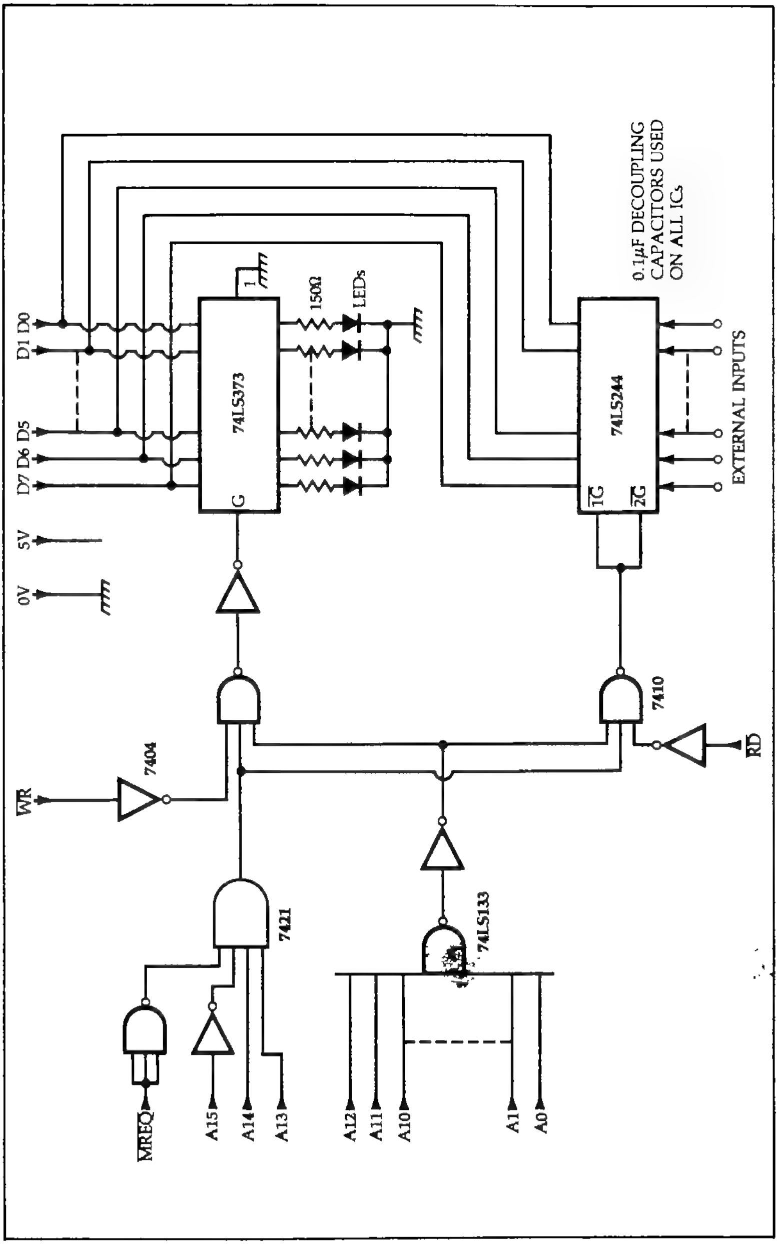


Fig. 9-17. One-byte memory mapped interface. All connecting leads should be as short as possible.

vided at the exposed edge connector (see Fig. 9-15 and Table 9-1).

To input and output data from or to your external hardware, for example using **PEEK** and **POKE,** appropriate address decoding logic must be provided to ensure that your hardware is memory mapped. Furthermore you have to include control signal logic for determining whether a read or write operation is to be implemented (Fig. 9-16). Your source of data must provide acceptable binary signals to the data bus for the memory-read cycle. However when the data bus is required by another device your input source connections must be isolated from the data bus. This can be achieved by using tristate data bus buffers. In contrast, when the machine writes data to the bus the device being written to must be capable of capturing the data during the memory write cycle. An octal latch is therefore suitable for this purpose.

Fig. 9-17 shows the logic diagram of a one-byte memory mapped interface. The address decoding logic maps this input/output port to the unique RAM address 32767 ($7FFF_{16}$). During the memory read cycle the octal data bus buffers (74LS244) are enabled by the $\overline{RD}$ signal thereby connecting the external inputs onto the data bus. During the memory write cycle the octal latch (74LS373) is enabled by the $\overline{WR}$ signal and the data is latched and displayed on the light-emitting diode (LED) array.

To read and display one byte of information from your external hardware to the computer use

PRINT PEEK 32767

To write one data byte to your external hardware use

POKE 32767, data byte.

Programs 7, 8, and 9 in Chapter 12 can be used to test the input/output interface shown in Fig. 9-17.

CHAPTER 10

The Heart of the Matter

How does the Z80A microprocessor implement instructions?

To answer this question we have to consider the functional block diagram of the Z80A shown in Fig. 10-1. The Z80A (MPU) addresses the system devices (RAM, ROM, and I/O) via a 16-bit *address bus,* and is thus capable of addressing 65 563 (2^{16}) unique memory locations in the range 0000 to FFFF. The 16 address lines may be conveniently partitioned into two bytes: address bits A0-A7 form the least significant byte (LS byte) of the address, and address bits A8-A15 form the most significant byte (MS byte) of the address. For example:

$$\underbrace{\underbrace{10001100}_{\text{MS byte}}\qquad\underbrace{00001110}_{\text{LS byte}}}_{\text{16-bit address bus} \equiv \text{8C0E}}$$

The MPU has an 8-bit bidirectional *data bus* through which it transmits data to, or receives data from, the RAM, ROM, and I/O devices. A data bus word will, for example, have the form

$$\underbrace{11100001}_{\text{8-bit data bus} \equiv \text{E1}}$$

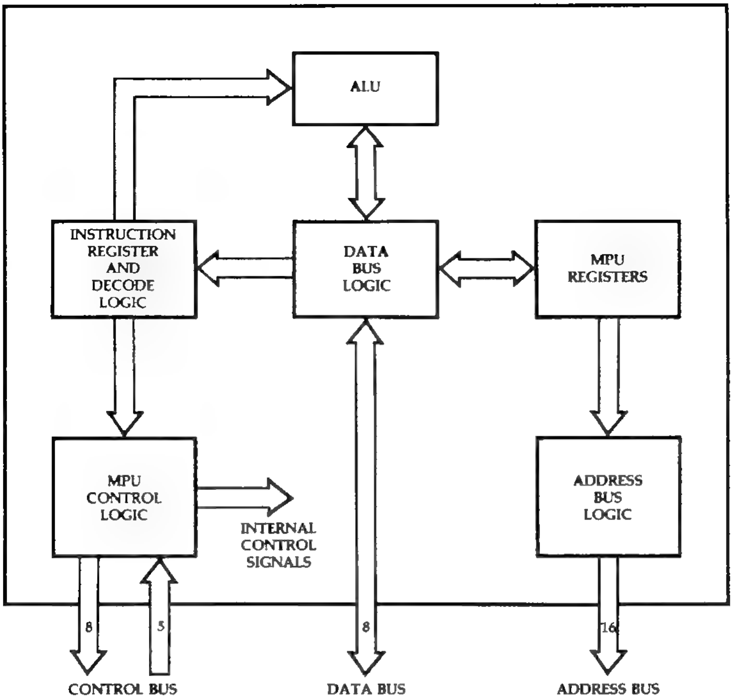


Fig. 10-1. Functional diagram of the Z80A.

The 16-bit address held in the MPU Program Counter register points to the next stored operation code (op-code) to be transferred to the MPU Instruction Register on receipt of the appropriate timing and control signals. The content of the Instruction Register is then decoded and the appropriate fetch, store, arithmetic, or logic operation is executed by the MPU.

To illustrate the principle of the MPU *fetch* and *execute* operation consider the instruction which immediately loads Register D in the MPU with data of value 6E. From the Instruction Set Summary given in Table 10-1 we see that the op-code for loading Register D using Immediate Addressing (LD r,n) is 00010110 (hexadecimal 16) which we will assume is stored in RAM location 17000 (hexadecimal 4268). The data word 6E will be stored in the RAM location immediately following the op-code, that is, at address 17001 (hexadecimal 4269).

Fig. 10-2 shows the timing diagram for this example and you may note that the complete instruction cycle consists of two machine cycles. The first machine cycle (M1) is used to fetch the op-code from memory for instruction decoding, and the second machine cycle (M2) is used to read the data byte into Register D. After completing this instruction the Program Counter will have been incremented to point to the next stored op-code, i.e. the Program Counter will point to memory location 17002 (hexadecimal 426A).

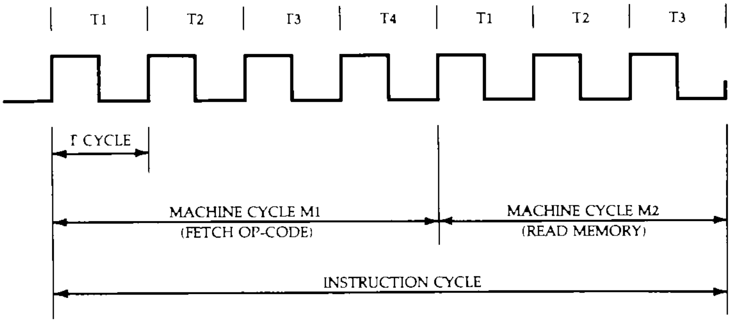


Fig. 10-2. Timing diagram for immediate addressing example.

The two M cycles are of different duration: M1 contains four T cycles and M2 contains three T cycles, where the duration of a T cycle corresponds to the MPU ϕ clock period. Obviously the number of T cycles per instruction cycle determines the time to fetch and execute an instruction, and this depends on the complexity of the instruction.

The waveform shown in Fig. 10-3 is an oscillogram of the MPU $\overline{\phi}$ clock which is provided on pin 6 of the exposed edge connector. The measured value of the T cycle period is 310.5 nanoseconds (310.5×10^{-9}s). Therefore in the above example involving seven T cycles the time to fetch and execute the instruction is 7×310.5 nanoseconds $= 2.1735$ microseconds (2.1735×10^{-6}s).

What are the functions of the Z80A input and output signals?

To enable you to connect peripheral hardware to your Timex Sinclair 1000/ZX81 (e.g. 16K RAM, printer, I/O devices)

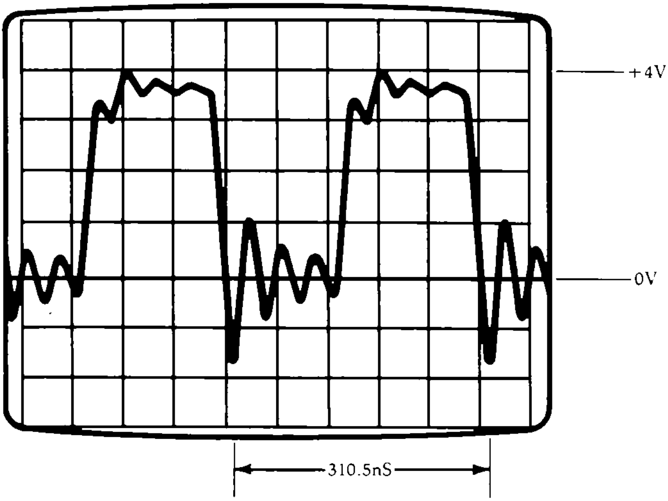


Fig. 10-3. MPU ϕ clock waveform.

the MPU pin connections are brought out to the exposed edge connector, as has already been mentioned. Only when you understand the function of each MPU signal will you be in a position to design and connect your own interface circuits. To assist you the pin assignment of the Z80A is given in Fig. 10-4 and the functions of the signals are summarized below.

The 40-pin dual-in-line NMOS Z80A is operated from a single +5V power supply. The externally generated, single-phase, TTL-level clock (ϕ) drives the MPU control and timing logic, and defines the T cycle period (Fig. 10-3).

The tristate, active-high, 8-bit input/output bidirectional data bus connections, D0 (least significant bit) to D7 (most significant bit), are used for data transfers between the MPU and memory and I/O devices. The tristate, active-high, 16-bit output unidirectional address bus connections, A0 (least significant bit) to A15 (most significant bit) are used for setting up the address for memory and I/O device data transfers.

The control bus consists of five input and eight output connections. The five inputs are:

1. $\overline{\text{WAIT}}$. This is an active-low input which makes the

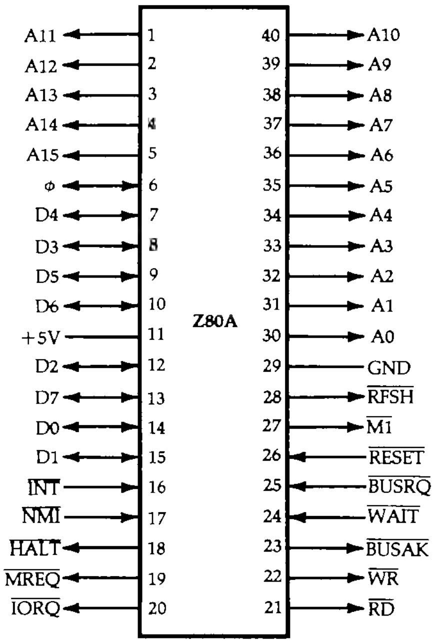


Fig. 10-4. Pinout diagram for Z80A MPU.

MPU enter *wait states.* These are ineffective T cycles inserted into a machine cycle and the MPU idles until the $\overline{WAIT}$ is set high. Therefore it is possible to control the MPU so that it will wait until memory or I/O devices are ready to make transfers along the data bus.

2. $\overline{RESET}$. This is the active-low input which sets the Program Counter to zero, disables the interrupt enable flip-flop, sets interrupt operation to mode 0, and clears Registers R and I.

3. $\overline{INT}$. This is the active-low interrupt request input, which responds to a signal generated by an I/O device when the interrupt enable flip-flop (IFF) is enabled, and when the $\overline{BUSRQ}$ signal is inactive. The response to an accepted $\overline{INT}$ input signal is determined by the specified interrupt mode code (Fig. 10-5).

4. $\overline{NMI}$. This is the active-low, negative edge triggered, nonmaskable interrupt input. It has a higher priority than $\overline{INT}$ and is implemented at the end of the current instruction irrespective of the state of the interrupt enable flip-flop

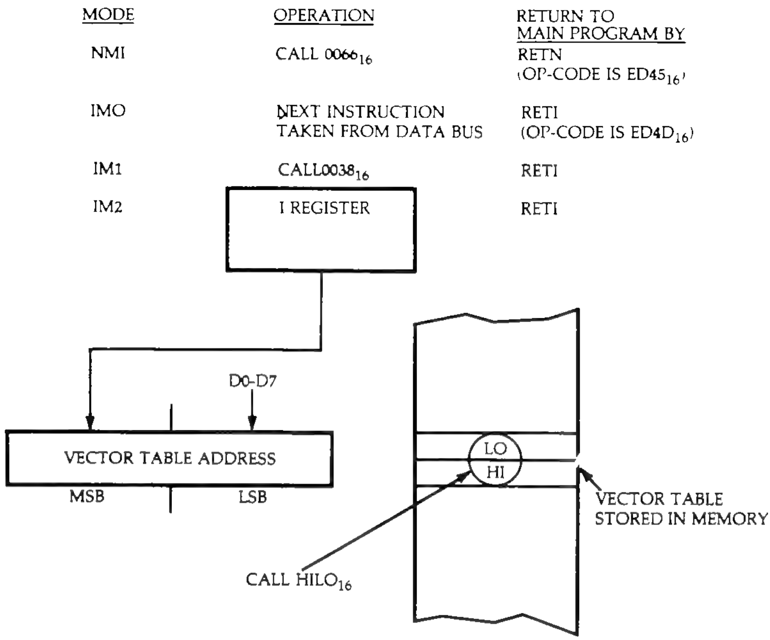


Fig. 10-5. Interrupt mode summary.

(IFF). The response to an acceptable $\overline{\text{NMI}}$ request is to make the MPU vector to memory location 102 (i.e. hexadecimal 0066). The $\overline{\text{NMI}}$ signal will not be accepted if $\overline{\text{WAIT}}$ or $\overline{\text{BUSRQ}}$ are active.

5. $\overline{\text{BUSRQ}}$. This is the active-low bus request input, and when enabled it sets the MPU address and data bus signals and the tristate control output signals to the high impedance state. It is used in cases when external devices control the buses. The high-impedance state is activated at the end of the machine cycle in which the $\overline{\text{BUSRQ}}$ signal is activated.

The eight output connections are:

1. $\overline{\text{BUSAK}}$. This is an active-low bus acknowledge output which, when active, indicates that the MPU address and data bus and tristate control bus signals are in the high impedance state.

2. $\overline{\text{HALT}}$. After executing a software Halt instruction, this output goes active low, and the MPU continues executing

NOPs (the instruction for no operation). It remains in this state until an interrupt signal ($\overline{NMI}$ or $\overline{INT}$) is received.

3. $\overline{RFSH}$. This is an active-low output which can be used with dynamic memories to refresh stored data. When the address bus lines A0 to A6 contain a refresh address this output is active.

4. $\overline{WR}$. When the data bus holds valid data this tristate active-low output indicates this condition to an addressed device and the data may be then written into the device.

5. $\overline{RD}$. When this tristate output is active low, it may be used by an addressed device to recognize that the MPU is ready to read data from the data bus.

6. $\overline{IORQ}$. This active-low tristate output indicates that a valid memory address is present on address lines A0 to A7.

7. $\overline{MREQ}$. This active-low tristate output indicates that a valid memory address is present on address lines A0 to A15.

8. $\overline{M1}$. This output will be active-low during machine cycle M1, i.e. for the period shown in Fig. 10-2.

Which Z80A registers are accessible to the programmer?

The MPU register configuration for the Z80A microprocessor is shown in Fig. 10-6. It contains 22 program-accessible internal registers.

There are two sets of general-purpose registers, each set containing six 8-bit registers, and there are two sets of accumulator and flag registers. The general-purpose 8-bit registers in each set may be concatenated (paired) to form 16-bit register pairs BC, DE and HL for the main register set, and B'C', D'E' and H'L' for the alternate register set.

The programmer, by using a single exchange instruction, can work with either the main register set or the alternate register set. This feature is useful where fast response to an interrupt is required, and in such cases the interrupt service routine can be programmed to substitute quickly the alternate registers for the main register set, and then switch back at the end of the interrupt routine.

The accumulator and flag register in the main register set or alternate register set are selected with a single exchange instruction. The 8-bit accumulator holds the result of arith-

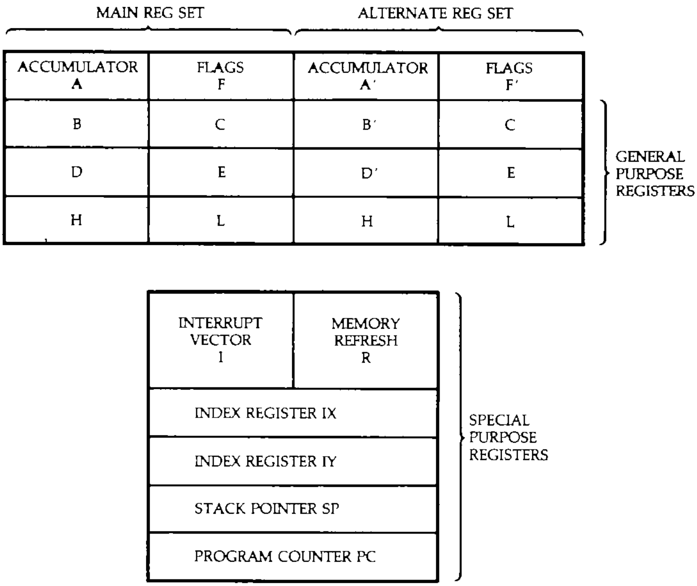


Fig. 10-6. Z80A program-accessible registers.

metic or logical operations and the flag register holds the states corresponding to the occurrence of specific events resulting from the execution of instructions (Fig. 10-7). For example, the carry flag C contains the highest order bit of the accumulator according to the instruction executed.

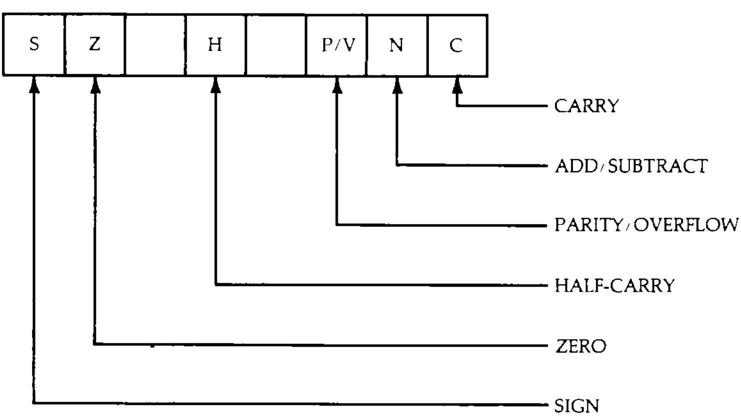


Fig. 10-7. Flag-register format.

The Z80A also contains four 16-bit registers (IX, IY, SP, and PC), an 8-bit I register, and a 7-bit R register. These are special purpose registers and they are described below.

The two 16-bit registers IX and IY are independent index registers which are used to hold 16-bit base addresses. These are used with indexed addressing mode instructions, whereby a two's complement signed integer (included as an additional byte in the instruction) specifies the displacement from the base address and this yields the address of the memory location to be accessed.

The 16-bit Stack Pointer (SP) contains the address of the current top of the stack, which is RAM that accepts or outputs data in a last-in first-out mode of operation. That is, data can be pushed onto the stack from specified MPU registers or popped off the stack to specified MPU registers.

The 16-bit Program Counter (PC) contains the memory address of the instruction currently being fetched from memory. Its content is changed in accordance with the requirements to fetch and execute the next instruction.

The 8-bit I register is the Interrupt Page Address Register. It is used when an interrupt is sensed by the MPU, and it stores the high-order eight bits of the vector table address. The interrupting device supplies the lower eight bits of the vector table address (Fig. 10-5).

The 7-bit Memory Refresh Register (R) is a counter which is automatically incremented after each M1 cycle (Fig. 10-2). It contains the address A0 to A6 that can be used to access dynamic memory locations, which are then refreshed by the $\overline{\text{RFSH}}$ Z80A output signal. This refresh operation does not interfere with normal MPU operation and it is transparent to the programmer.

How is the Z80A Instruction Set summarized?

It is useful to the user to have the 158 different types of Z80A instructions logically arranged in groups as shown in Tables 10-1 to 10-11. For the instructions in each table the assembly language mnemonic, the symbolic operation, the status of the Flag Register bits, the op-code (binary or hex form) and relevant storage, and timing information are provided. Comments, notes, and flag notation are also included.

We shall see in Chapter 11 how to use some of these instructions when writing machine-code programs.

Table 10-1. Z80A 8-Bit Load Instructions

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments r, s	Reg.
LD r, s	r ← s	•	•	X	•	X	•	•	•	01 r s		1	1	4	r, s	Reg.
LD r, n	r ← n	•	•	X	•	X	•	•	•	00 r 110		2	2	7	000	B
										← n →					001	C
LD r, (HL)	r ← (HL)	•	•	X	•	X	•	•	•	01 r 110		1	2	7	010	D
LD r, (IX+d)	r ← (IX+d)	•	•	X	•	X	•	•	•	11 011 101	DD	3	5	19	011	E
										01 r 110					100	H
										← d →					101	L
LD r, (IY+d)	r ← (IY+d)	•	•	X	•	X	•	•	•	11 111 101	FD	3	5	19	111	A
										01 r 110						
										← d →						
LD (HL), r	(HL) ← r	•	•	X	•	X	•	•	•	01 110 r		1	2	7		
LD (IX+d), r	(IX+d) ← r	•	•	X	•	X	•	•	•	11 011 101	DD	3	5	19		
										01 110 r						
										← d →						
LD (IY+d), r	(IY+d) ← r	•	•	X	•	X	•	•	•	11 111 101	FD	3	5	19		
										01 110 r						
										← d →						
LD (HL), n	(HL) ← n	•	•	X	•	X	•	•	•	00 110 110	36	2	3	10		
										← n →						
LD (IX+d), n	(IX+d) ← n	•	•	X	•	X	•	•	•	11 011 101	DD	4	5	19		
										00 110 110	36					
										← d →						
										← n →						
LD (IY+d), n	(IY+d) ← n	•	•	X	•	X	•	•	•	11 111 101	FD	4	5	19		
										00 110 110	36					
										← d →						
										← n →						
LD A, (BC)	A ← (BC)	•	•	X	•	X	•	•	•	00 001 010	0A	1	2	7		
LD A, (DE)	A ← (DE)	•	•	X	•	X	•	•	•	00 011 010	1A	1	2	7		
LD A, (nn)	A ← (nn)	•	•	X	•	X	•	•	•	00 111 010	3A	3	4	13		
										← n →						
										← n →						
LD (BC), A	(BC) ← A	•	•	X	•	X	•	•	•	00 000 010	02	1	2	7		
LD (DE), A	(DE) ← A	•	•	X	•	X	•	•	•	00 010 010	12	1	2	7		
LD (nn), A	(nn) ← A	•	•	X	•	X	•	•	•	00 110 010	32	3	4	13		
										← n →						
										← n →						
LD A, I	A ← I	↕	↕	X	0	X	IFF	0	•	11 101 101	ED	2	2	9		
										01 010 111	57					
LD A, R	A ← R	↕	↕	X	0	X	IFF	0	•	11 101 101	ED	2	2	9		
										01 011 111	5F					
LD I, A	I ← A	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	9		
										01 000 111	47					
LD R, A	R ← A	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	9		
										01 001 111	4F					

Notes: r, s means any of the registers A, B, C, D, E, H, L
IFF the content of the interrupt enable flip-flop (IFF) is copied into the P/V flag

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 10-2. Z80A 16-Bit Load Instructions

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
LD dd, nn	dd ← nn	•	•	X	•	X	•	•	•	00 dd0 001		3	3	10	dd Pair
										← n →					00 BC
										← n →					01 DE
LD IX, nn	IX ← nn	•	•	X	•	X	•	•	•	11 011 101	DD	4	4	14	10 HL
										00 100 001	21				11 SP
										← n →					
										← n →					
LD IY, nn	IY ← nn	•	•	X	•	X	•	•	•	11 111 101	FD	4	4	14	
										00 100 001	21				
										← n →					
										← n →					
LD HL, (nn)	H ← (nn+1)	•	•	X	•	X	•	•	•	00 101 010	2A	3	5	16	
	L ← (nn)									← n →					
										← n →					
LD dd, (nn)	dd_H ← (nn+1)	•	•	X	•	X	•	•	•	11 101 101	ED	4	6	20	
	dd_L ← (nn)									01 dd1 011					
										← n →					
										← n →					
LD IX, (nn)	IX_H ← (nn+1)	•	•	X	•	X	•	•	•	11 011 101	DD	4	6	20	
	IX_L ← (nn)									00 101 010	2A				
										← n →					
										← n →					
LD IY, (nn)	IY_H ← (nn+1)	•	•	X	•	X	•	•	•	11 111 101	FD	4	6	20	
	IY_L ← (nn)									00 101 010	2A				
										← n →					
										← n →					
LD (nn), HL	(nn+1) ← H	•	•	X	•	X	•	•	•	00 100 010	22	3	5	16	
	(nn) ← L									← n →					
										← n →					
LD (nn), dd	(nn+1) ← dd_H	•	•	X	•	X	•	•	•	11 101 101	ED	4	6	20	
	(nn) ← dd_L									01 dd0 011					
										← n →					
										← n →					
LD (nn), IX	(nn+1) ← IX_H	•	•	X	•	X	•	•	•	11 011 101	DD	4	6	20	
	(nn) ← IX_L									00 100 010	22				
										← n →					
										← n →					
LD (nn), IY	(nn+1) ← IY_H	•	•	X	•	X	•	•	•	11 111 101	FD	4	6	20	
	(nn) ← IY_L									00 100 010	22				
										← n →					
										← n →					
LD SP, HL	SP ← HL	•	•	X	•	X	•	•	•	11 111 001	F9	1	1	6	
LD SP, IX	SP ← IX	•	•	X	•	X	•	•	•	11 011 101	DD	2	2	10	
										11 111 001	F9				
LD SP, IY	SP ← IY	•	•	X	•	X	•	•	•	11 111 101	FD	2	2	10	
										11 111 001	F9				qq Pair
PUSH qq	(SP-2) ← qq_L	•	•	X	•	X	•	•	•	11 qq0 101		1	3	11	00 BC
	(SP-1) ← qq_H														01 DE
PUSH IX	(SP-2) ← IX_L	•	•	X	•	X	•	•	•	11 011 101	DD	2	4	15	10 HL
	(SP-1) ← IX_H									11 100 101	E5				11 AF
PUSH IY	(SP-2) ← IY_L	•	•	X	•	X	•	•	•	11 111 101	FD	2	4	15	
	(SP-1) ← IY_H									11 100 101	E5				
POP qq	qq_H ← (SP+1)	•	•	X	•	X	•	•	•	11 qq0 001		1	3	10	
	qq_L ← (SP)														
POP IX	IX_H ← (SP+1)	•	•	X	•	X	•	•	•	11 011 101	DD	2	4	14	
	IX_L ← (SP)									11 100 001	E1				
POP IY	IY_H ← (SP+1)	•	•	X	•	X	•	•	•	11 111 101	FD	2	4	14	
	IY_L ← (SP)									11 100 001	E1				

Notes: dd is any of the register pairs BC, DE, HL, SP
qq is any of the register pairs AF, BC, DE, HL
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively.
e.g. BC_L = C, AF_H = A

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ flag is affected according to the result of the operation.

Table 10-3. Z80A Exchange Group and Block Transfer and Search Instructions

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
EX DE, HL	DE↔HL	•	•	X	•	X	•	•	•	11 101 011	EB	1	1	4	
EX AF, AF'	AF↔AF'	•	•	X	•	X	•	•	•	00 001 000	08	1	1	4	
EXX	BC↔BC' DE↔DE' HL↔HL'	•	•	X	•	X	•	•	•	11 011 001	D9	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H ↔(SP+1) L ↔(SP)	•	•	X	•	X	•	•	•	11 100 011	E3	1	5	19	
EX (SP), IX	IX_H ↔(SP+1) IX_L ↔(SP)	•	•	X	•	X	•	•	•	11 011 101 11 100 011	DD E3	2	6	23	
EX (SP), IY	IY_H ↔(SP+1) IY_L ↔(SP)	•	•	X	•	X	•	•	•	11 111 101 11 100 011	FD E3	2	6	23	
LDI	(DE)←(HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	•	•	X	0	X	① ↕	0	•	11 101 101 10 100 000	ED A0	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE)←(HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 Repeat until BC = 0	•	•	X	0	X	0	0	•	11 101 101 10 110 000	ED B0	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE)←(HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	•	•	X	0	X	① ↕	0	•	11 101 101 10 101 000	ED A8	2	4	16	
LDDR	(DE)←(HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 Repeat until BC = 0	•	•	X	0	X	0	0	•	11 101 101 10 111 000	ED B8	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A – (HL) HL ← HL+1 BC ← BC-1	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 100 001	ED A1	2	4	16	
CPIR	A – (HL) HL ← HL+1 BC ← BC-1 Repeat until A = (HL) or BC = 0	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 110 001	ED B1	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A – (HL) HL ← HL-1 BC ← BC-1	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 101 001	ED A9	2	4	16	
CPDR	A – (HL) HL ← HL-1 BC ← BC-1 Repeat until A = (HL) or BC = 0	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 111 001	ED B9	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 10-4. Z80A 8-Bit Arithmetic and Logical Instructions

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments	
ADD A, r	A ← A + r	↕	↕	X	↕	X	V	0	↕	10 [000] r		1	1	4	r	Reg.
ADD A, n	A ← A + n	↕	↕	X	↕	X	V	0	↕	11 [000] 110		2	2	7	000	B
										← n →					001	C
															010	D
ADD A, (HL)	A ← A+(HL)	↕	↕	X	↕	X	V	0	↕	10 [000] 110		1	2	7	011	E
ADD A, (IX+d)	A ← A+(IX+d)	↕	↕	X	↕	X	V	0	↕	11 011 101	DD	3	5	19	100	H
										10 [000] 110					101	L
										← d →					111	A
ADD A, (IY+d)	A ← A+(IY+d)	↕	↕	X	↕	X	V	0	↕	11 111 101	FD	3	5	19		
										10 [000] 110						
										← d →						
ADC A, s	A ← A+s+CY	↕	↕	X	↕	X	V	0	↕	[001]					s is any of r, n,	
SUB s	A ← A - s	↕	↕	X	↕	X	V	1	↕	[010]					(HL), (IX+d),	
SBC A, s	A ← A - s - CY	↕	↕	X	↕	X	V	1	↕	[011]					(IY+d) as shown for	
AND s	A ← A ∧ s	↕	↕	X	1	X	P	0	0	[100]					ADD instruction.	
OR s	A ← A ∨ s	↕	↕	X	0	X	P	0	0	[110]					The indicated bits	
XOR s	A ← A ⊕ s	↕	↕	X	0	X	P	0	0	[101]					replace the [000] in	
CP s	A - s	↕	↕	X	↕	X	V	1	↕	[111]					the ADD set above.	
INC r	r ← r + 1	↕	↕	X	↕	X	V	0	•	00 r [100]		1	1	4		
INC (HL)	(HL) ← (HL)+1	↕	↕	X	↕	X	V	0	•	00 110 [100]		1	3	11		
INC (IX+d)	(IX+d) ← (IX+d)+1	↕	↕	X	↕	X	V	0	•	11 011 101	DD	3	6	23		
										00 110 [100]						
										← d →						
INC (IY+d)	(IY+d) ← (IY+d)+1	↕	↕	X	↕	X	V	0	•	11 111 101	FD	3	6	23		
										00 110 [100]						
										← d →						
DEC s	s ← s - 1	↕	↕	X	↕	X	V	1	•	[101]					s is any of r, (HL), (IX+d), (IY+d) as shown for INC. DEC same format and states as INC. Replace [100] with [101] in OP Code.	

Notes: The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation. Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow, P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

Table 10-5. Z80A General Purpose Arithmetic and MPU Control Instructions

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
DAA	Converts acc. content into packed BCD following add or subtract with packed BCD operands	↕	↕	X	↕	X	P	•	↕	00 100 111	27	1	1	4	Decimal adjust accumulator
CPL	$A \leftarrow \overline{A}$	•	•	X	1	X	•	1	•	00 101 111	2F	1	1	4	Complement accumulator (One's complement)
NEG	$A \leftarrow \overline{A}+1$	↕	↕	X	↕	X	V	1	↕	11 101 101 01 000 100	ED 44	2	2	8	Negate acc. (two's complement)
CCF	$CY \leftarrow \overline{CY}$	•	•	X	X	X	•	0	↕	00 111 111	3F	1	1	4	Complement carry flag
SCF	$CY \leftarrow 1$	•	•	X	0	X	•	0	1	00 110 111	37	1	1	4	Set carry flag
NOP	No operation	•	•	X	•	X	•	•	•	00 000 000	00	1	1	4	
HALT	CPU halted	•	•	X	•	X	•	•	•	01 110 110	76	1	1	4	
DI*	$IFF \leftarrow 0$	•	•	X	•	X	•	•	•	11 110 011	F3	1	1	4	
EI*	$IFF \leftarrow 1$	•	•	X	•	X	•	•	•	11 111 011	FB	1	1	4	
IM 0	Set interrupt mode 0	•	•	X	•	X	•	•	•	11 101 101 01 000 110	ED 46	2	2	8	
IM 1	Set interrupt mode 1	•	•	X	•	X	•	•	•	11 101 101 01 010 110	ED 56	2	2	8	
IM 2	Set interrupt mode 2	•	•	X	•	X	•	•	•	11 101 101 01 011 110	ED 5E	2	2	8	

Notes: IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

*Interrupts are not sampled at the end of EI or DI

Table 10-6. Z80A 16-Bit Arithmetic Instructions

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments	
ADD HL, ss	HL ← HL+ss	•	•	X	X	X	•	0	↕	00 ss1 001		1	3	11	ss	Reg.
															00	BC
ADC HL, ss	HL ← HL+ss+CY	↕	↕	X	X	X	V	0	↕	11 101 101	ED	2	4	15	01	DE
										01 ss1 010					10	HL
															11	SP
SBC HL, ss	HL ← HL-ss-CY	↕	↕	X	X	X	V	1	↕	11 101 101	ED	2	4	15		
										01 ss0 010						
ADD IX, pp	IX ← IX + pp	•	•	X	X	X	•	0	↕	11 011 101	DD	2	4	15	pp	Reg.
										00 pp1 001					00	BC
															01	DE
															10	IX
															11	SP
ADD IY, rr	IY ← IY + rr	•	•	X	X	X	•	0	↕	11 111 101	FD	2	4	15	rr	Reg.
										00 rr1 001					00	BC
															01	DE
															10	IY
															11	SP
INC ss	ss ← ss + 1	•	•	X	•	X	•	•	•	00 ss0 011		1	1	6		
INC IX	IX ← IX + 1	•	•	X	•	X	•	•	•	11 011 101	DD	2	2	10		
										00 100 011	23					
INC IY	IY ← IY + 1	•	•	X	•	X	•	•	•	11 111 101	FD	2	2	10		
										00 100 011	23					
DEC ss	ss ← ss - 1	•	•	X	•	X	•	•	•	00 ss1 011		1	1	6		
DEC IX	IX ← IX - 1	•	•	X	•	X	•	•	•	11 011 101	DD	2	2	10		
										00 101 011	2B					
DEC IY	IY ← IY - 1	•	•	X	•	X	•	•	•	11 111 101	FD	2	2	10		
										00 101 011	2B					

Notes: ss is any of the register pairs BC, DE, HL, SP
pp is any of the register pairs BC, DE, IX, SP
rr is any of the register pairs BC, DE, IY, SP.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

Table 10-7. Z80A Rotate and Shift Instructions

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
RLCA	CY ← 7 ← 0 ← (A)	•	•	X	0	X	•	0	↕	00 000 111	07	1	1	4	Rotate left circular accumulator
RLA	← CY ← 7 ← 0 ← (A)	•	•	X	0	X	•	0	↕	00 010 111	17	1	1	4	Rotate left accumulator
RRCA	→ 7 → 0 → CY (A)	•	•	X	0	X	•	0	↕	00 001 111	0F	1	1	4	Rotate right circular accumulator
RRA	→ 7 → 0 → CY → (A)	•	•	X	0	X	•	0	↕	00 011 111	1F	1	1	4	Rotate right accumulator
RLC r		↕	↕	X	0	X	P	0	↕	11 001 011 00 000 r	CB	2	2	8	Rotate left circular register r
RLC (HL)		↕	↕	X	0	X	P	0	↕	11 001 011 00 000 110	CB	2	4	15	r Reg. 000 B 001 C
RLC (IX+d)	CY ← 7 ← 0 ← r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	11 011 101 11 001 011 – d – 00 000 110	DD CB	4	6	23	010 D 011 E 100 H 101 L 111 A
RLC (IY+d)		↕	↕	X	0	X	P	0	↕	11 111 101 11 001 011 – d – 00 000 110	FD CB	4	6	23	
RL s	← CY ← 7 ← 0 ← s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	010					Instruction format and states are as shown for RLC's. To form new Op-Code replace 000 of RLC's with shown code
RRC s	→ 7 → 0 → CY s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	001					
RR s	→ 7 → 0 → CY → s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	011					
SLA s	CY ← 7 ← 0 ← 0 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	100					
SRA s	→ 7 → 0 → CY s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	101					
SRL s	0 → 7 → 0 → CY s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	111					
RLD	A 7-4 3-0 7-4 3-0 (HL)	↕	↕	X	0	X	P	0	•	11 101 101 01 101 111	ED 6F	2	5	18	Rotate digit left and right between the accumulator and location (HL).
RRD	A 7-4 3-0 7-4 3-0 (HL)	↕	↕	X	0	X	P	0	•	11 101 101 01 100 111	ED 67	2	5	18	The content of the upper half of the accumulator is unaffected

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 10-8. Z80A Bit Set, Reset, and Test Instructions

Mnemonic	Symbolic Operation	Flags: S	Z		H		P/V	N	C	Op-Code: 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments	
BIT b, r	$Z \leftarrow \overline{r}_b$	X	↕	X	1	X	X	0	•	11 001 011	CB	2	2	8	r	Reg.
										01 b r					000	B
BIT b, (HL)	$Z \leftarrow \overline{(HL)}_b$	X	↕	X	1	X	X	0	•	11 001 011	CB	2	3	12	001	C
										01 b 110					010	D
BIT b, $(IX+d)_b$	$Z \leftarrow \overline{(IX+d)}_b$	X	↕	X	1	X	X	0	•	11 011 101	DD	4	5	20	011	E
										11 001 011	CB				100	H
										– d –					101	L
										01 b 110					111	A
															b	Bit Tested
BIT b, $(IY+d)_b$	$Z \leftarrow \overline{(IY+d)}_b$	X	↕	X	1	X	X	0	•	11 111 101	FD	4	5	20	000	0
										11 001 011	CB				001	1
										– d –					010	2
										01 b 110					011	3
															100	4
															101	5
															110	6
															111	7
SET b, r	$r_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 001 011	CB	2	2	8		
										$\boxed{11}$ b r						
SET b, (HL)	$(HL)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 001 011	CB	2	4	15		
										$\boxed{11}$ b 110						
SET b, (IX+d)	$(IX+d)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 011 101	DD	4	6	23		
										11 001 011	CB					
										– d –						
										$\boxed{11}$ b 110						
SET b, (IY+d)	$(IY+d)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 111 101	FD	4	6	23		
										11 001 011	CB					
										– d –						
										$\boxed{11}$ b 110						
RES b, s	$s_b \leftarrow 0$ $s \equiv r$, (HL), (IX+d), (IY+d)	•	•	X	•	X	•	•	•	$\boxed{10}$					To form new Op-Code replace $\boxed{11}$ of SET b, s with $\boxed{10}$. Flags and time states for SET instruction	

Notes: The notation s_b indicates bit b (0 to 7) or location s.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 10-9. Z80A Jump Instructions

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
JP nn	PC ← nn	•	•	X	•	X	•	•	•	11 000 011	C3	3	3	10	
										← n →					
										← n →					
JP cc, nn	If condition cc is true PC ← nn, otherwise continue	•	•	X	•	X	•	•	•	11 cc 010		3	3	10	
										← n →					
										← n →					
JR e	PC ← PC + e	•	•	X	•	X	•	•	•	00 011 000	18	2	3	12	
										← e-2 →					
JR C, e	If C = 0, continue	•	•	X	•	X	•	•	•	00 111 000	38	2	2	7	If condition not met
										← e-2 →					
	If C = 1, PC ← PC+e											2	3	12	If condition is met
JR NC, e	If C = 1, continue	•	•	X	•	X	•	•	•	00 110 000	30	2	2	7	If condition not met
										← e-2 →					
	If C = 0, PC ← PC+e											2	3	12	If condition is met
JR Z, e	If Z = 0 continue	•	•	X	•	X	•	•	•	00 101 000	28	2	2	7	If condition not met
										← e-2 →					
	If Z = 1, PC ← PC+e											2	3	12	If condition is met
JR NZ, e	If Z = 1, continue	•	•	X	•	X	•	•	•	00 100 000	20	2	2	7	If condition not met
										← e-2 →					
	If Z = 0, PC ← PC+e											2	3	12	If condition is met
JP (HL)	PC ← HL	•	•	X	•	X	•	•	•	11 101 001	E9	1	1	4	
JP (IX)	PC ← IX	•	•	X	•	X	•	•	•	11 011 101	DD	2	2	8	
										11 101 001	E9				
JP (IY)	PC ← IY	•	•	X	•	X	•	•	•	11 111 101	FD	2	2	8	
										11 101 001	E9				
DJNZ, e	B ← B-1 If B = 0, continue	•	•	X	•	X	•	•	•	00 010 000	10	2	2	8	If B = 0
										← e-2 →					
	If B ≠ 0, PC ← PC+e											2	3	13	If B ≠ 0

cc	Condition
000	NZ non zero
001	Z zero
010	NC non carry
011	C carry
100	PO parity odd
101	PE parity even
110	P sign positive
111	M sign negative

Notes: e represents the extension in the relative addressing mode.

e is a signed two's complement number in the range <126, 129>

e-2 in the op-code provides an effective address of pc+e as PC is incremented by 2 prior to the addition of e.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ↕ = flag is affected according to the result of the operation.

Table 10-10. Z80A Call and Return Instructions

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	76	543	210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
CALL nn	$(SP-1) \leftarrow PC_H$	•	•	X	•	X	•	•	•	11	001	101	CD	3	5	17	
	$(SP-2) \leftarrow PC_L$									←	n	→					
	$PC \leftarrow nn$									←	n	→					
CALL cc, nn	If condition	•	•	X	•	X	•	•	•	11	cc	100		3	3	10	If cc is false
	cc is false									←	n	→					
	continue,									←	n	→		3	5	17	If cc is true
	otherwise																
	same as																
	CALL nn																
RET	$PC_L \leftarrow (SP)$	•	•	X	•	X	•	•	•	11	001	001	C9	1	3	10	
	$PC_H \leftarrow (SP+1)$																
RET cc	If condition	•	•	X	•	X	•	•	•	11	cc	000		1	1	5	If cc is false
	cc is false																
	continue,													1	3	11	If cc is true
	otherwise																
	same as																
	RET																
RETI	Return from	•	•	X	•	X	•	•	•	11	101	101	ED	2	4	14	
	interrupt									01	001	101	4D				
RETN[1]	Return from	•	•	X	•	X	•	•	•	11	101	101	ED	2	4	14	
	non maskable									01	000	101	45				
	interrupt																
RST p	$(SP-1) \leftarrow PC_H$	•	•	X	•	X	•	•	•	11	t	111		1	3	11	
	$(SP-2) \leftarrow PC_L$																
	$PC_H \leftarrow 0$																
	$PC_L \leftarrow p$																

cc	Condition	
000	NZ	non zero
001	Z	zero
010	NC	non carry
011	C	carry
100	PO	parity odd
101	PE	parity even
110	P	sign positive
111	M	sign negative

t	p
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

[1] RETN loads $IFF_2 \rightarrow IFF_1$

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 10-11. Z80A Input and Output Instructions

Mnemonic	Symbolic Operation	Flags: S	Z		H		P/V	N	C	Op-Code: 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
IN A, (n)	A ← (n)	•	•	X	•	X	•	•	•	11 011 011	DB	2	3	11	n to $A_0 \sim A_7$
										– n –					Acc to $A_8 \sim A_{15}$
IN r, (C)	r ← (C)	↕	↕	X	↕	X	P	0	•	11 101 101	ED	2	3	12	C to $A_0 \sim A_7$
	if r = 110 only									01 r 000					B to $A_8 \sim A_{15}$
	the flags will														
	be affected														
INI	(HL) ← (C)	X	↕ [1]	X	X	X	X	1	X	11 101 101	ED	2	4	16	C to $A_0 \sim A_7$
	B ← B – 1									10 100 010	A2				B to $A_8 \sim A_{15}$
	HL ← HL + 1														
INIR	(HL) ← (C)	X	1	X	X	X	X	1	X	11 101 101	ED	2	5	21	C to $A_0 \sim A_7$
	B ← B – 1									10 110 010	B2		(If B ≠ 0)		B to $A_8 \sim A_{15}$
	HL ← HL + 1											2	4	16	
	Repeat until												(If B = 0)		
	B = 0														
IND	(HL) ← (C)	X	↕ [1]	X	X	X	X	1	X	11 101 101	ED	2	4	16	C to $A_0 \sim A_7$
	B ← B – 1									10 101 010	AA				B to $A_8 \sim A_{15}$
	HL ← HL – 1														
INDR	(HL) ← (C)	X	1	X	X	X	X	1	X	11 101 101	ED	2	5	21	C to $A_0 \sim A_7$
	B ← B – 1									10 111 010	BA		(If B ≠ 0)		B to $A_8 \sim A_{15}$
	HL ← HL – 1											2	4	16	
	Repeat until												(If B = 0)		
	B = 0														
OUT (n), A	(n) ← A	•	•	X	•	X	•	•	•	11 010 011	D3	2	3	11	n to $A_0 \sim A_7$
															Acc to $A_8 \sim A_{15}$
OUT (C), r	(C) ← r	•	•	X	•	X	•	•	•	11 101 101	ED	2	3	12	C to $A_0 \sim A_7$
										01 r 001					B to $A_8 \sim A_{15}$
OUTI	B ← B – 1	X	↕ [1]	X	X	X	X	1	X	11 101 101	ED	2	4	16	C to $A_0 \sim A_7$
	(C) ← (HL)									10 100 011	A3				B to $A_8 \sim A_{15}$
	HL ← HL + 1														
OTIR	B ← B – 1	X	1	X	X	X	X	1	X	11 101 101	ED	2	5	21	C to $A_0 \sim A_7$
	(C) ← (HL)									10 110 011	B3		(If B ≠ 0)		B to $A_8 \sim A_{15}$
	HL ← HL + 1											2	4	16	
	Repeat until												(If B = 0)		
	B = 0														
OUTD	(C) ← (HL)	X	↕ [1]	X	X	X	X	1	X	11 101 101	ED	2	4	16	C to $A_0 \sim A_7$
	B ← B – 1									10 101 011	AB				B to $A_8 \sim A_{15}$
	HL ← HL – 1														
OTDR	(C) ← (HL)	X	1	X	X	X	X	1	X	11 101 101	ED	2	5	21	C to $A_0 \sim A_7$
	B ← B – 1									10 111 011	BB		(If B ≠ 0)		B to $A_8 \sim A_{15}$
	HL ← HL – 1											2	4	16	
	Repeat until												(If B = 0)		
	B = 0														

Notes: ① If the result of B – 1 is zero the Z flag is set, otherwise it is reset.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

CHAPTER 11

Cracking Machine Code

Why use machine code?

When programs are written in BASIC the statements, functions and numbers have to be stored and subsequently interpreted by the Timex Sinclair 1000/ZX81 operating system. The BASIC commands are interpreted so that the Z80A microprocessor can fetch and execute the instructions. This is necessary because the Z80A can only work with machine code and therefore only fetches and executes instructions supplied in this form, as described in Chapter 10.

The interpretation process takes a finite time to implement, and if this can be eliminated by writing programs in machine code rather than BASIC, then a significant time saving is achieved. This is an important consideration in applications involving graphics or control of external peripheral devices.

The number of memory bytes required for a program written in BASIC is much larger than the number required for the equivalent machine-code program. Consequently a significant saving in memory requirements is achieved by writing programs in machine code, leaving storage space for more programs.

How do I write machine code programs?

In Chapter 10 we described how a machine code instruction is fetched and executed by the Z80A microprocessor. Furthermore, we discussed the accessible registers of the

Z80A (Fig. 10-6) and summarized its instruction set (see Tables 10-1 to 10-11). However, to use this information effectively in writing machine code programs, you need to understand the various Z80A addressing modes, which are described in this section.

Register Addressing

This mode of addressing has a single-byte op-code which defines the MPU registers involved. In Table 10-1 the first instruction, LD r,s uses this form of addressing. For example, if we wish to copy (load) the content of register C into register B the op-code is

01 register code for B register code for C

01 000 001

4 1

Thus the op-code 41, stored in a single byte of memory, implements the required operation. Note that as each register of the Z80A has a different 3-bit code, you can copy the content of any register into another register using the appropriate op-code.

Register Indirect Addressing

The content of a general-purpose register pair or a 16-bit special-purpose register forms the memory address of the data byte to be accessed. For example, to load register A with the accessed data in the memory address specified by the content of the DE register pair the instruction LD A, (DE) is used; i.e. from Table 10-1 it is seen that op-code 1A implements the instruction.

In contrast, as a further example of this form of addressing, consider the instruction POP IX. This loads the low byte of the index register IX with the content of the data held at the memory address given by the content of the Stack Pointer, SP, and loads the high byte of the index register with the data held at the memory address given by the Stack Pointer plus one. From Table 10-2 we can see that the op-code for the two-byte instruction is:

DD
E1

Immediate Addressing

In this mode of addressing the data byte *immediately* follows the op-code. For example the instruction of the form LD r,n loads register r with the byte n. Thus to preset register E with data equal to F8 we see from Table 10-1 that the instruction is:

1E	Op-Code
F8	Data

Immediate Extended Addressing

This form of addressing can be used to preset register pairs and the 16-bit special-purpose registers. For example, using Table 10-2, we can see that the instruction

DD 21	Op-code
FF	Data LO
00	Data HI

loads the Index Register IX with the data 00FF, and the instruction

01	Op-code
BA	Data LO
AB	Data HI

loads the register pair BC with the data ABBA.

Extended Addressing

The last two bytes of the instruction are used to specify the address of the memory location for the data to be used with the instruction, or they contain an address used with a jump instruction. For example, the instruction LD A,(nn) loads accumulator A with the content of the memory location having the address nn. From Table 10-1, we see that to load accumulator A with the content of memory location 1982 we use the instruction

3A	Op-code
82	Address LO
19	Address HI

A further example is the instruction of the form LD dd,(nn), which loads register pair dd (see the 2-bit code in Table

10-2) with data from memory addresses nn and nn + 1, where the content of the memory byte addressed by nn is loaded into the LO register of the pair and the content of the memory byte addressed by nn + 1 is loaded into the HI register of the pair. Thus to load register pair BC with the data held in memory locations 1982 and 1983 the required instruction is:

ED 4B	Op-code	
82	LO	Address of data for register C
19	HI	

A straightforward form of jump instruction that uses this form of addressing is the Unconditional Jump, JP, nn (Table 10-9). For example, the instruction

C3	Op-code
21	Address LO
C4	Address HI

will make the program jump to memory address C421 thereby accessing the first byte of the next instruction.

In contrast, the jump instruction can be made conditional upon the state of a specified flag condition by using the JP cc,nn instruction given in Table 10-9. For example, the instruction

CA	Op-code
2B	Address LO
14	Address HI

will cause the program to jump to memory location 142B *if* the result of the last instruction was zero. If not, the jump instruction is ignored and the next consecutive instruction in the program is executed.

Modified Page Zero Addressing

The eight one-byte restart instructions, RST p, given in Table 10-10, use modified page zero addressing to direct program control to one of the eight predetermined addresses, which have a HI address byte of 00, hence the name given to this addressing mode. The instruction to gain access to the predetermined memory location requires only the single-byte op-code, without the need for any additional

address bytes. This results in a saving of memory space and time. This is a useful feature to use for accessing frequently used subroutines.

The Timex Sinclair 1000/ZX81 ROM uses the eight restart instructions in its monitor routines. For example, the RST to memory address 0008 is used for the start address of the Error Report Handling routine. As shown in Table 10-10 the op-code CF is used to access 0008. To use this facility in machine-code programs to generate your own report codes use the op-code CF followed by the data byte required to define the alphanumeric report code character to be displayed. The data byte used is obtained by subtracting 1D from its hex code, as defined in Table 7-1. For example, to display report code P (hex code 35) you would use the instruction

CF	Op-code
18	Data byte (35-1D)

Implied Addressing

Instructions which operate with the content of a specific register are known as implied addressing mode instructions. For example, this is the case for arithmetic and logic operations on contents of accumulator A. This form of addressing is illustrated in the following examples:

1. To increment the content of a selected register we can use the instruction INC r. For instance, to increment the content of register B we can use the op-code, 04, obtained from Table 10-4.

2. To form the one's complement of the content of accumulator A use the instruction CPL. The corresponding op-code is 2F (Table 10-5).

3. To shift the content of register H one place to the left, with a 0 transferred into the least significant bit of H and the most significant bit of H transferring into the carry bit, you can use the instruction SLA s. The required op-code obtained from Table 10-7 is:

CB
24

The comment in Table 10-7, that the required instruction format and states are as shown for the RLC r instructions, de-

fines the first byte as being CB and the second byte as 00100110, i.e. 24.

Bit Addressing

It is possible to address a single bit within a selected register or RAM location and set or reset the bit according to the specified instruction.

The following examples illustrate this form of addressing. Note that the bits in a byte are numbered 0, 1, 2, . . . 6 and 7, where bit 0 is the *least* significant bit and bit 7 is the *most* significant bit.

1. The instruction BIT b, r (given in Table 10-8) will set the Z flag equal to the complement of the logic state of the addressed bit, b, in the selected r register. For example, to set the Z flag equal to the complement of bit 3 in the L register the required op-code is:

CB
5D

2. The instruction RES b, (HL) is used to reset bit b in the RAM location addressed by the content of the HL register pair (Table 10-8). For example, the instruction

CB
AE (10 101 110)

will reset bit 5 in the RAM location addressed by the content of the HL register at the time of executing the instruction.

Indexed Addressing

The Z80A microprocessor uses this form of memory addressing to access the desired data byte to be used with the instruction. It sets up the required memory address by adding the content of the selected index register (IX or IY) to a *displacement byte* (d) with the index register conten' and displacement byte unchanged when the instruction is executed.

The displacement byte is a two's complement, 8-bit binary number (see Chapter 3) and therefore it is possible for this byte to have a value in the range +127(01111111) to −128(10000000) inclusive. Consequently memory locations

are addressed using (IX + d) or (IY + d), where $-128 \leqslant d \leqslant 127$.

The instruction LD r, (IX + d) given in Table 10-1 will now be used to illustrate this mode of addressing. For this example suppose that the content of the index register IX is 0200 and that it is required that the data held in memory location 01FB is to be copied (loaded) into register E. The required instruction (see Table 10-1) is:

DD	
5E	(01 011 110)
FC	(11111011 = −5 = 01FB − 0200)

Relative Addressing

This form of addressing mode applies to some of the Z80A jump instructions, which are formed by the appropriate op-code followed by a two's complement displacement byte denoted by e − 2.

The microprocessor achieves the desired change (jump) in program control by modifying the content of the Program Counter (PC) thereby enabling access to the next desired program instruction.

The two's complement binary number equal to 'e' is added to the memory address of the jump instruction op-code (held in PC) to achieve the desired modification to (PC). Since e − 2 is a two's complement 8-bit binary number representing numbers in the range $-128 \leqslant (e-2) \leqslant 127$, then it follows that program control may be made to jump to any location in the range −126 to 129 from the address of the jump instruction op-code. This means that because program control does not jump to a specified memory address it may be possible to easily relocate the program in another part of memory. It is better, therefore, to use relative addressing mode instructions rather than extended addressing mode instructions. The following example uses the instruction JR e (given in Table 10-9) to illustrate this mode of addressing.

Suppose that the required instruction op-code (18) is stored in memory location 4AFC and a program control jump to memory location 4B02 is required, then the necessary machine code instruction (obtained from Table 10-9) is:

18	Op-code
04	e – 2 (e = 6; 4AFC + 6 = 4B02)

In our consideration of the addressing modes we have shown the machine codes for selected instructions. When you store these types of instructions in sequential memory locations, you have created a machine-code program, which the microprocessor runs by fetching and executing each instruction in turn.

As an illustration of a machine-code program, listed below are the codes for a program which loads accumulator A with the hexadecimal number 8D, then copies the content of A into register B, then forms the one's complement of the content of A and finally copies the content of A into register C.

3E 8D	Op-code Data	LD A,8D	(Table 10-1)
47	Op-code	LD B,A	(Table 10-1)
2F	Op-code	CPL	(Table 10-5)
4F	Op-code	LD C,A	(Table 10-1)

Implementation of these four instructions results in (A) = 72, (B) = 8D and (C) = 72, where ( ) indicates register content.

Where can I store machine code programs?

There are two places in memory where you can store a machine-code program. It can either be stored in the same area as your BASIC program or in a separate area.

To include machine code within a BASIC program you can use an appropriate **REM** statement. However, from a practical point of view, the choice of where to place the **REM** statement is governed by the need to know the address of the first byte of the machine-code program. When the **REM** is the first line of your BASIC program, the address of the first machine-code byte will be 16514, because the address of the first byte for any BASIC program is 16509 (Fig. 9-14) and the line number and **REM** token require four and one memory locations respectively, i.e. 16509 + 4 + 1 = 16514. If you place the **REM** statement elsewhere in your program you will have to determine the address in RAM that

can hold the first byte of the machine-code program, and this is not an easy task.

When using the **REM** statement in the first line of your BASIC program the number of characters between the **REM** token and **ENTER** corresponds to the number of bytes that can be used for machine code. For example, to enter the three-byte machine-code program

04	INC B
0D	DEC C
C9	RET

in the **REM** statement, we can convert the three bytes of machine code to their corresponding characters by using the Character Set Summary (Table 7-1). So

04	converts to ▖
0D	converts to $
C9	converts to TAN

and we then enter the program line

1 **REM**▖$TAN

The machine code program will be in your machine with the first op-code (04) being located at memory address 16514.

You should note that in Table 7-1 some of the hex codes do not have an equivalent keyboard character (43_{16} to $6F_{16}$ and $7A_{16}$ to $7D_{16}$ and $C3_{16}$), so when using this method to input any of these op-codes, dummy characters should be entered at the appropriate places in the **REM** statement to reserve memory bytes, and then subsequently the corresponding memory locations can be **POKE**d with the required decimal equivalent codes. Furthermore, because the BASIC monitor automatically controls the occurrence of the **K** cursor, it may not be possible to enter a machine-code byte corresponding to a keyword character code, in the desired place in the **REM** statement. In such cases you will again have to reserve memory bytes by inserting dummy characters, which can subsequently be replaced with the required decimal equivalent codes using **POKE** statements. This process forms the basis of the method described next.

An alternative method of entering machine-code programs uses a two-pass operation. In the first pass the **REM**

statement is used to reserve the required memory space for the machine-code program, and in the second pass the machine-code bytes are converted to their decimal equivalent code and **POKE**d into the appropriate memory locations. To enter the three-byte program using this method, the first pass operation is to enter the program line

1 **REM** BBB

where the three characters BBB are included to reserve three bytes of memory for the machine code. These bytes of the machine code must now be converted to their decimal equivalents (Table 7-1), i.e.

04	converts to 04
0D	converts to 13
C9	converts to 201,

and subsequently **POKE**d into memory address 16514, 16515 and 16516. Note that on completing this two-pass operation your **REM** statement line will be listed as

1 **REM** ◧ $TAN

which is identical to the line entered by the previous method.

When your machine code is included in a **REM** statement it is part of a BASIC program and is subject to all the BASIC commands, e.g. **EDIT, LIST, SAVE, NEW** (which erases the entire program), etc.

If you do not wish to have your machine-code program erased by **NEW** you should store it in memory above the system variable known as RAMTOP, where RAMTOP defines the upper limit of RAM available for the BASIC programs, and its LO and HI bytes are held in addresses 16388 and 16389 respectively. For example, to set RAMTOP equal to 17325 the required LO and HI bytes to be **POKE**d are 173 and 67 respectively, that is:

HI*265 + LO = RAMTOP

Once you have defined RAMTOP you can **POKE** your machine-code program in memory locations having addresses above this value.

As an example let us see how to enter the machine-code program

```
04
0D
C9
```

at memory locations 17325, 17326 and 17327, where RAMTOP is to be defined as 17325. The first step is to **POKE** the required LO and HI bytes of RAMTOP, i.e. enter

```
POKE 16388, 173
POKE 16389, 67
NEW
```

Next we convert the machine-code bytes to their decimal equivalents (Table 7-1) and **POKE** them with the required addresses, i.e. enter

```
POKE 17325, 04
POKE 17326, 13
POKE 17327, 201
```

The tedium of calculating the LO and HI bytes of RAMTOP, then **POKE**ing these values into the RAMTOP system variable locations, followed by the subsequent conversion of machine-code bytes to their decimal equivalent and then **POKE**ing them into memory locations above RAMTOP, can be eliminated by using program 1 in Chapter 12.

The main disadvantage of storing machine-code above RAMTOP is that it cannot be saved on cassette tape. However, it may be easily reloaded using Program 1 in Chapter 12.

How can I link a machine code program to a BASIC program?

The link is established by using the **USR** function. The memory address of the first byte of machine code is the number used after the **USR** function. The hexadecimal equivalent of this number is loaded into the BC register pair of the Z80A and, after executing the last machine-code instruction, i.e. the essential return instruction (C9), the content of the BC register pair returns to the **USR** function.

For example the machine-code program discussed in the previous question, i.e. above RAMTOP set equal to 17325, is

Memory Address	Op-code	Comment
17325	Ø4	INC B
17326	ØD	DEC C
17327	C9	RET

Once this program has been entered it can be accessed and run using

PRINT USR 17325

followed by **ENTER.** The displayed result is 17580 which is the correct answer because, on entering the machine-code program

$$(B) = 67 \text{ and } (C) = 173$$

(recall that 17325 = 67 × 256 + 173), and then after executing the first instruction (INCB)

$$(B) = 68 \text{ and } (C) = 173$$

and then after executing the second instruction (DEC C)

$$(B) = 68 \text{ and } (C) = 172$$

and then after the return (RET) instruction, the **USR** function has the value 68 × 256 + 172 = 17580.

It is worth noting that the **USR** function is often used in a line of a BASIC program to access the machine-code program. For example, if you use

18 **LET** A = **USR** 17325

then, after executing the machine-code program above, A is set equal to 17580.

Note also that a return instruction (RET) must be included as the last op-code in the machine-code program. Otherwise it is quite possible that your machine will fetch and execute the codes which exist in the RAM locations following the end of your machine-code program and, since these are unspecified, a CRASH condition usually results. That is, the Timex Sinclair 1000/ZX81 operating system loses control and makes the microcomputer incapable of doing anything useful. You will recognize this condition because the tv display will become permanently blank or it will be showing irregular patterns similar to those observed when saving a program on cassette tape. Unfortunately, the only way to

restore control after a crash is momentarily to disconnect the 9V dc power supply, which of course results in the loss of your program in RAM.

How can I use some of the existing BASIC routines within machine-code programs?

Your machine-code program can access the BASIC routines in the ROM by including the CALL instruction with the address of the routine to be executed. You may recall that we discussed use of the "Error Report Routine" (RST 08) when we described modified page zero addressing earlier in this chapter.

The following are a useful selection of routines which have some influence on the tv display, and they can be included in your machine-code programs. They can be particularly useful for graphics applications.

***SLOW Routine:* CALL 0F28**

Access to this routine is gained by using the following three bytes of machine code:

CD	CALL 0F28
28	LO byte of Routine address
0F	HI byte of Routine address

***FAST Routine:* CALL 0F20**

Access to this routine is gained by using the following three bytes of machine code:

CD	CALL 0F20
20	LO byte of Routine address
0F	HI byte of Routine address

***CLS Routine:* CALL 0A2A**

The following three bytes of machine code will provide access:

CD	CALL 0A2A
2A	LO byte of Routine address
0A	HI byte of Routine address

***SCROLL Routine:* CALL ØCØE**

Access to this routine is gained by using the following three bytes of machine code:

CD	CALL ØCØE
ØE	LO byte of Routine address
ØC	HI byte of Routine address

See Program 2 in Chapter 12 for an example of using this routine.

***PRINT AT Routine:* CALL Ø8F5**

In Fig. 7-1 we showed that the tv screen is a grid of "character squares," and that the position of a square is referenced by the row and column numbers. The "pointer" to a screen grid square can be initialized in machine-code programs using the PRINT AT Routine.

Before calling the PRINT AT Routine it is necessary to load register C with the column reference number and register B with the row reference number.

The following six bytes of machine code will initialize the PRINT AT pointer:

Ø1	LD BC, nn
n	column number hexadecimal code
n	row number hexadecimal code
CD	CALL Ø8F5
F5	LO byte of Routine address
Ø8	HI byte of Routine address

***PRINT Character Routine:* RST 1Ø**

To print a single character on the tv screen the A register must be loaded with the hexadecimal code corresponding to the desired character (Table 7-1) and then the RST 1Ø routine is called. This is achieved using three bytes of machine code as follows:

3E	LD A,n
n	hexadecimal code for desired character
D7	RST 1Ø

Note that if you wish the character to be printed in a particular screen grid square, then the screen grid pointer (con-

tent of BC register pair) must be initialized (see PRINT AT Routine above) immediately before the three machine-code bytes of the PRINT Character Routine. See Program 3 in Chapter 12 for an illustrative example.

***PRINT String Routine:* CALL 0B6B**

This routine is used to access and print on the tv screen a string of characters held in successive memory locations. Before calling this routine it is necessary to load the DE register pair with the memory address of the location which holds the first character code in the *look-up table.* Also before the routine is called it is necessary to load the BC register pair with a hexadecimal number correpsonding to the number of characters in the string.

The following nine bytes of machine code will implement this routine.

11	LD DE, nn
n	LO byte of address of first character code
n	HI byte of address of first character code
01	LD BC, nn
n	LO byte of number of character codes
n	HI byte of number of character codes
CD	CALL 0B6B
6B	LO byte of Routine address
0B	HI byte of Routine address

Note that if you wish the string to be printed at a position on the screen starting in a particular screen grid square, then the screen grid pointer must be initialized immediately before the PRINT String Routine (see the PRINT AT Routine above). Program 4 in Chapter 12 gives an example of the use of this routine.

***PRINT Positive Integer (0-9999) Routine:* CALL 0AAB**

To print a positive decimal integer in the range 0 to 9999 $(0\text{-}270F)_{16}$ on the tv screen, the HL register pair must be loaded with the hexadecimal equivalent of the decimal number before this routine is called.

The six bytes of machine code to achieve this desired print operation are:

21	LD HL, nn
n	LO byte of number to be printed
n	HI byte of number to be printed.
CD	CALL 0AAB
AB	LO byte of Routine address
0A	HI byte of Routine address

Again, if you wish to have the number printed at a position on the screen starting in a particular grid square, then the screen pointer must be initialized immediately before this routine (see the PRINT AT Routine). Program 5 in Chapter 12 gives an illustrative example.

***PLOT/UNPLOT Routine:* CALL 0BB2**

In Fig. 7-4 we showed that the tv screen is a grid of "pixel squares," and that a pixel is referenced by an *x* and *y* number. The *x* and *y* parameters must be loaded into the BC register pair before this routine is called. Furthermore, before calling this routine the system variable T__ADDR (memory location 16432) must be initialized to the value 9B to **PLOT** the selected pixel, or be initialized to the value A0 to **UNPLOT** the selected pixel.

The following eleven bytes of machine code may be used to **PLOT** or **UNPLOT** a selected pixel.

01	LD BC, nn
n	x parameter value $(0-63)_{10} = (0-3F)_{16}$
n	y parameter value $(0-43)_{10} = (0-2B)_{16}$
3E	LD A, n
n	9B for **PLOT** or A0 for **UNPLOT**
32	LD(4030),A $(16432)_{10} = (4030)_{16}$
30	LO byte of address for T__ADDR
40	HI byte of address for T__ADDR
CD	CALL 0BB2
B2	LO byte of Routine address
0B	HI byte of Routine address

See Program 6 in Chapter 12 for an illustrative example.

CHAPTER 12

More Programs To Try

1. Program to load machine code above RAMTOP

This program requires you to input, as a decimal number, the value of RAMTOP greater than 16850 (i.e. above this program), which is then stored in the defined memory locations for this system variable. You can then input each byte of your machine-code program as two hexadecimal characters and these are stored in successive memory locations starting at the address defined by RAMTOP. When the last byte of your machine-code program has been entered, you must enter Z to terminate the loading operation.

If you find in entering a lengthy machine code program that the screen becomes full and consequently displays report code 5, you can make the Timex Sinclair 1000/ZX81 continue by entering **CONT** followed by **ENTER**.

```
 5 PRINT "INPUT RAMTOP VALUE"
15 INPUT RT
25 PRINT RT
35 LET X = INT (RT/256)
45 LET Y = RT - 256*X
55 POKE 16388,Y
65 POKE 16389,X
75 INPUT H$
77 PRINT H$
85 IF H$ = "Z" THEN NEW
```

```
 95 POKE RT,16*CODE H$ + CODE H$(2) - 476
115 LET RT = RT + 1
125 GOTO 75
```

Note that the **NEW** statement in line 85 is necessary to enable the new value of RAMTOP to be established, and to permit access to machine code programs using the **USR** function. Because this **NEW** statement is included it deletes the loading program after the machine code program has been entered, and therefore it is advisable to have this loading program saved on cassette tape.

2. Program to verify the SCROLL Routine

First, load and run the following BASIC program:

```
10 PRINT AT 21,15;"*"
15 FOR I = 1 TO 20
35 SCROLL
40 PAUSE 10
45 NEXT I
```

You should observe that the asterisk scrolls vertically up the screen. Now change the program to:

```
 1 REM LN :£TAN
10 PRINT AT 21,15;"*"
15 FOR I = 1 TO 20
35 LET J = USR 16514
40 PAUSE 10
45 NEXT I
```

Run this program and observe that its operation is identical to the first program. Hence you will see that the machine-code program contained in the **REM** statement corresponds to the **SCROLL** statement.

3. Program to verify the PRINT Character and the PRINT AT Routines

This is a machine-code program that can be used to print a character in a specified screen grid square. As an example, if we want an "inverse + " to be printed in square 2,30 then the required machine code program is:

```
01  LD BC, 021E
1E  column number in hex
02  row number in hex
CD  CALL 08F5
F5
08
3E  LD A,95
95  character code
D7  CALL RST 10
C9  RET
```

Use program 1 in this chapter to set RAMTOP equal to 17300, and then enter the above machine codes. Run the machine code program by entering

RUN USR 17300 **ENTER**

Observe that the "inverse +" is printed at square 2,30 on the screen.

4. Program to verify the PRINT String Routine

This is a machine-code program that can be used to print a string of characters. As an example, suppose that it is required that the message: NOW SWITCH OFF be printed in row 21, starting in column 9, then the required machine-code program is:

```
01  LD BC, 1509
09  column number in hex
15  row number in hex
CD  CALL 08F5
F5
08
11  LD DE, 43A4
A4
43
01  LD BC, 000E
0E
00
CD  CALL 0B6B
6B
0B
C9  RET
```

```
33  N
34  O
3C  W
00  SPACE
38  S
3C  W
2E  I
39  T
28  C
2D  H
00  SPACE
34  O
2B  F
2B  F
```

look-up table

Use program 1 in this chapter to set RAMTOP equal to 17300, and then enter the above machine codes. Run the machine-code program by entering:

```
220 FOR N = 1 TO 2
225 LET J = USR 17300
235 NEXT N
245 STOP
```

followed by **RUN** 220, then **ENTER.**

Don't obey the displayed message unless you want to lose your program!

5. Program to verify the PRINT Positive Integer Routine

This is a machine-code program that can be used to print a positive integer in the range 0 to 9999. As an example, suppose that it is required to print 5051 (13BB in hex) in row 12, starting in column 18, then the required machine-code program is:

```
01  LD BC,0C12
12  column number in hex
0C  row number in hex
CD  CALL 08F5
F5
08
21  LD HL,13BB
```

```
BB
13
CD  CALL 0AAB
AB
0A
C9  RET
```

Use program 1 in this chapter to set RAMTOP equal to 17300, and then enter the above machine codes. Run the machine-code program by entering:

225 **LET** J = **USR** 17300

followed by **RUN** 225, then **ENTER.**

You will see the number displayed.

6. Program to verify the PLOT/UNPLOT Routine

This is a machine-code program that can be used to **PLOT** or **UNPLOT** a pixel. As an example, suppose that it is required to **PLOT** pixel 20,8 (see Fig. 7-4), then the required machine-code program is:

```
01  LD BC,0814
14
08
3E  LD A,9B
9B
32  LD(4030),A
30
40
CD  CALL 0BB2
B2
0B
C9  RET
```

Use program 1 in this chapter to set RAMTOP equal to 17300, and then enter the above machine codes. Run the machine-code program by entering:

225 **LET** J = **USR** 17300

followed by **RUN** 225, then **ENTER.** The pixel will then be displayed.

To investigate the **UNPLOT** Routine change the data byte, 9B, in memory location 17304 to A0(160_{10}), i.e.

POKE 17304, 160 followed by **ENTER.**

Now enter the following additional BASIC program lines

```
135 FOR I = 6 TO 9
145 FOR K = 19 TO 22
155 PLOT K,I
165 NEXT K
175 NEXT I
```

These are included to black in a small area of the screen, and then when the **UNPLOT** Routine is executed for a selected pixel in this black area the action of UNPLOTting will be clearly seen. Run the program by entering

RUN 135, then **ENTER.**

7. Program to test the Input Port

This program periodically reads in the data byte provided by the external user hardware (see Fig. 9-17) and displays the decimal equivalent of the 8-bit number on the tv screen. The program is

```
 5 PRINT PEEK 32767
25 PAUSE 100
35 CLS
45 GOTO 5
```

8. Program to test the Output Port

This program outputs a data byte to the LED array shown in Fig. 9-17, such that each LED is illuminated sequentially for a short period of time as determined by the **PAUSE** statement.

```
 5 LET X = 1
55 POKE 32767,X
65 PAUSE 50
75 IF X = 128 THEN GOTO 5
85 LET X = 2*X
95 GOTO 55
```

9. Security System Monitor

This program uses the input port shown in Fig. 9-17 to monitor the state of the voltage levels on four data input

lines, which connect via the tristate input buffers to the data bus connections D7, D6, D5, and D4. If any of these lines are at logic 1, i.e., 5V, the program detects this and outputs an appropriate message to the tv display (see the **PRINT** statements in the program listing below).

```
  5 LET X = PEEK 32767
 10 PRINT "SYSTEM CHECK"
 15 LET J = X
 25 LET N = 2*J
 35 IF N < 255 THEN GOTO 75
 45 PRINT "DOOR 1 OPEN"
 55 LET N1 = (N - 256)/2
 65 GOTO 80
 75 LET N1 = N/2
 80 LET N2 = 2*N1
 85 IF N2 < 127 THEN GOTO 135
 95 PRINT "DOOR 2 OPEN"
115 LET N3 = (N2 - 128)/2
125 GOTO 145
135 LET N3 = N2/2
145 LET N4 = 2*N3
155 IF N4 < 63 THEN GOTO 195
165 PRINT "DOOR 3 OPEN"
175 LET N5 = (N4 - 64)/2
185 GOTO 205
195 LET N5 = N4/2
205 LET N6 = 2*N5
215 IF N6 < 31 THEN GOTO 245
225 PRINT "DOOR 4 OPEN"
245 PAUSE 100
255 CLS
265 GOTO 5
```

Typically this program could be used to monitor the security of four doors in your home, where the 5V input level may be derived from a microswitch when the door is open. When the door is closed a 0V input will be present.

If you wish to test the program without providing the actual signals to the edge connector, then simply change line 5 in the above program to

```
5 INPUT X
```

This then requires you to input 128 to simulate door 1 open (i.e. 10000000), 64 for door 2 open, 32 for door 3 open, and 16 for door 4 open, or a number which is the sum of any of the above; for example X = 96 will cause the display to indicate that doors 2 and 3 are open.

Appendix

Glossary of Terms

ACCESS TIME—The time from the receipt of an address by a memory element to the time for the data from the addressed element to appear at the output.
ACCUMULATOR—A register that may be used as the source of one operand and the destination of results from device operations.
ACTIVE-HIGH—Logic level 1 is the active state.
ACTIVE-LOW—Logic level 0 is the active state.
ADDRESS—The coded location of one byte of memory.
ADDRESSING MODES—These are the modes specifying the memory addresses or microprocessor registers to be used with an instruction.
ALPHANUMERIC—Refers to alphabetic and numeric characters.
ARCHITECTURE—The structure of a system.
ARITHMETIC LOGIC UNIT (ALU)—An element that can perform several arithmetic and logic functions.
ARITHMETIC SHIFT—A shift operation (left or right) in which the value of the sign bit is maintained.
ARRAY—A set of variables with each variable identified by a single character representing the array, and subscript numbers representing position within the array, e.g. B(3,2), B(4,4).
ASSEMBLER—A computer program that is used to translate mnemonic instructions into their binary equivalent codes and assign memory locations for data and instructions.

ASSEMBLY LANGUAGE—A language in which mnemonic instructions, labels and names are used. These can be translated by an assembler into a machine-code program.

ASYNCHRONOUS OPERATION—Operation without using a timing reference.

BACKING STORE—A large-capacity store such as a cassette tape.

BASIC—*B*eginners *A*ll-purpose *S*ymbolic *I*nstruction *C*ode. A high-level language, popular for use with microcomputers.

BAUD RATE—The serial rate of transmission expressed in bits per second.

BENCHMARK PROGRAM—A specimen program that is used to compare and evaluate microprocessors.

BIDIRECTIONAL BUS—A bus along which signals may be sent in either direction.

BINARY—A number system with base or radix 2.

BINARY CODED DECIMAL (BCD)—A code in which each decimal digit is coded using four weighted binary digits.

BIT—A *bi*nary digi*t*.

BREAKPOINT—A user-specified temporary program halt used in program debugging.

BUG—An error in a program.

BUS—Parallel conductors connecting two or more devices.

BUS CONTENTION—The situation when two or more devices are simultaneously attempting to place data on a data bus.

BYTE—The group of eight bits considered by the microprocessor as a binary word.

CARRY BIT—The bit used to indicate the occurrence of a carry from the most significant bit in a word.

CHIP—The substrate of a single integrated circuit.

CLEAR—An input to a device that resets the states to 0.

CLOCK—A periodic timing signal used to control a system.

CMOS DEVICES—*C*omplementary *M*etal *O*xide *S*emiconductor logic elements constructed from *n*- or *p*-channel field-effect transistors to provide lower power consumption and high noise immunity devices.

CONDITIONAL JUMP—An instruction that causes a jump to a different part of the program when a given condition is true.

COUNTER—A device that changes state after the application of each clock pulse. Normally its output indicates the total number of clock pulses received (up to its capacity).

CRASH—Loss of control of a program.

CURRENT LOOP—An interface connection that normally uses 20 mA current in the loop to indicate the logic 1 and zero current to indicate logic 0.

CYCLE TIME—The time interval for a set of regular operations to be repeated.

DEBOUNCE—The conversion of mechanical contact bounce into a clean transition between the two logic states.

DEBUG—Removal of programming errors.

DECIMAL ADJUST—An operation to convert a binary result into a binary coded decimal result.

DELAY TIME—The time between demand signal and appearance of a corresponding output response.

DEMULTIPLEXER—A device that directs a time-shared input signal to several outputs in order to separate the channels.

DUMP—Transfer to a backing store.

DYNAMIC MEMORY—A memory that slowly loses its contents and therefore needs refreshing.

EAROM—*E*lectrically *A*lterable *R*ead *O*nly *M*emory.

EEROM / **E^{2}PROM**—*E*lectrically *E*rasable *R*ead *O*nly *M*emory.

EPROM—*E*rasable *P*rogrammable *R*ead *O*nly *M*emory.

FIELD PROGRAMMABLE ROM—A read only memory that can be programmed by the user.

FIRMWARE—Microprograms implemented in read only memory.

FLAG—A flip-flop that is normally set to logic 1 after the occurrence of a specified event.

FLOWCHART—A graphical representation of a set of computer instructions.

GLITCH—An unwanted electrical noise pulse.

HARDWARE—The physical devices constituting a computer.

HEXADECIMAL—A number system with radix 16. It uses the alphanumeric characters 0 to 9 and A to F.

HIGH-LEVEL LANGUAGE—A computer language in which the statements represent procedures as opposed to single machine instructions.

INDEX REGISTER—A microprocessor register used for memory address modification.

INSTRUCTION—A group of bits to specify a microprocessor operation.

INSTRUCTION CYCLE—The cycle of fetching, decoding, and executing an instruction.

INSTRUCTION EXECUTION TIME—The time required to fetch, decode and execute an instruction.

INSTRUCTION SET—The set of instructions that can be interpreted by the microprocessor.

INTEGER—A positive or negative whole number or zero.

INTERRUPT—A microprocessor input that temporarily transfers control from the main program to a separate interrupt routine.

INTERRUPT MASKING—A technique that permits the microprocessor to specify if interrupts will be accepted.

INTERRUPT ROUTINE—A program that is implemented in response to an interrupt signal.

INVERSE VIDEO—Display of a character as white on black.

K OF MEMORY—1024 bytes of memory.

LABEL—A name given to an instruction or statement in a program in order to identify it.

LATCH—A temporary storage element, usually a flip-flop.

LOOP—A sequence of instructions that a computer repeats.

MACHINE CODE—The language that a computer can interpret directly.

MACHINE CYCLE—The basic microprocessor cycle. It is the time required to fetch data from memory or to execute a one-byte instruction.

MASK—(a) A bit pattern that separates one or several bits from a group of bits. (b) A photographic plate used in integrated circuit fabrication to define the diffusion patterns.

MASKABLE INTERRUPT—An interrupt that can be disabled by the system.

MEMORY—An element that can store logic bits.

MEMORY MAP—A graphical method of illustrating designated memory sections.

MICROCOMPUTER—A microprocessor, memory and interface elements assembled to form a computer.

MICROINSTRUCTION—One of the organized sequence of control signals that form instructions at the control level.

MICROPROCESSOR—The central processing unit of a microcomputer.

MODEM—A modulator/demodulator element that uses a carrier frequency in order to permit communication on a high frequency channel.

MONITOR—A simple operating system that permits the user to enter and run programs.

MOS DEVICES—Semiconductor elements that use field-effect transistors manufactured using *M*etal *O*xide *S*ilicon.

MULTIPLEXER—An element that selects one of several inputs and places it on a time shared output.

MULTIPROCESSING—Using more than one microprocessor in a single system.

NESTED LOOPS—Instruction loops within instruction loops.

NIBBLE—A group of four bits.

NMOS—*N*-channel *M*etal *O*xide *S*emiconductor.

NONDESTRUCTIVE READOUT—The content of the element can be read without changing the current.

NONMASKABLE INTERRUPT—An interrupt that cannot be disabled by the system.

NONVOLATILE MEMORY—A memory that maintains its content even when the power is removed.

NULL STRING—A string containing no characters.

NUMERIC VARIABLE—A variable which has a name, and can have a numerical value or numerical expression assigned to it.

OFFSET—A number that is added to another number to form an effective address.

ONE'S COMPLEMENT—A bit-by-bit complement of a binary number.

OP-CODE—The part of a machine code instruction used to specify the operation to be undertaken during the next cycle.

PARITY—A one-bit code that is added to a word to make

the total number of one-bits in the word even (even parity) or odd (odd parity).

PARITY BIT—A status bit that is normally set to logic 1 if the last operation gave a result with (a) even parity, if even parity is used, or (b) odd parity if odd parity is used.

PEEK—An instruction in BASIC used to examine the content of a memory byte.

PIXEL—A picture element.

PMOS—*P*-channel *M*etal *O*xide *S*emiconductor.

POINTER—A register or memory location that contains an address.

POKE—An instruction in BASIC used to write a data byte into a memory location.

POLLING—The successive examination of the state of peripherals.

POP—Remove an operand from the stack.

PRIORITY INTERRUPTS—Interrupts that can be serviced before others, or may interrupt other interrupt routines.

PROGRAM—A sequence of instructions correctly ordered to implement a specific task.

PROGRAM COUNTER—A register that holds the address of the next instruction to be executed.

PROGRAMMED INPUT/OUTPUT—Input/output operation implemented under program control.

PROM—*P*rogrammable *R*ead *O*nly *M*emory.

PUSH—Put an operand onto the stack.

RAM—A memory that can be read and written into. It is referred to as a *R*andom *A*ccess *M*emory.

REAL-TIME CLOCK—An element that interrupts a microprocessor at regular time intervals.

REFRESH—The process of restoring the content of a dynamic memory.

REGISTER—A group of memory cells used to store words within a microprocessor.

ROM—*R*ead *O*nly *M*emory.

SECOND SOURCE—A manufacturer who supplies a device originated by another manufacturer.

SIGN BIT—The most significant bit of data word that indicates the sign of the data. Logic 0 is used to indicate a positive number and logic 1 is used to indicate a negative number.

SIGNAL CONDITIONING—Changing a signal to make it compatible with a specific device.

SOFTWARE—Computer programs.

SOFTWARE INTERRUPT—An instruction that makes a program jump to a specific address.

STACK—A group of RAM memory elements that are normally accessed in a last-in, first-out way.

STACK POINTER—A register used to address the next available stack location.

STATIC MEMORY—A memory that does not require refreshing.

STRING—A set of characters enclosed within a program.

STRING VARIABLE—A variable, consisting of a single alphabetic character followed by **$**, to which a string can be assigned.

SUBROUTINE—A subprogram that can be entered from more than one place in a main program.

SYNCHRONOUS OPERATION—Operating at regular intervals of time with respect to a reference time.

SYNTAX—The rules of statement structure in a programming language.

TERMINAL—An input/output device used to communicate with a microprocessor system.

TRISTATE—Logic outputs with three possible output levels, namely low, high, and high impedance.

TTL—*T*ransistor *T*ransistor *L*ogic. This is a very popular bipolar technology used in integrated circuits.

TTL-COMPATIBLE DEVICES—These use voltage and current levels within the TTL range and do not need level shifting for interfacing to TTL devices.

TWO'S COMPLEMENT—The one's complement of a binary number plus one.

VOLATILE MEMORY—A memory that loses its content when the power is removed.

WORD—The group of bits that a microprocessor can manipulate.

WORD LENGTH—The number of bits in a microprocessor word.

Index

TIMEX SINCLAIR 1000™/ZX81 USER'S HANDBOOK

The Timex Sinclair 1000/ZX81 is a fascinating machine that has brought the power of the computer within everyone's reach. However, the user is often bewildered by the many facilities offered by his computer, and after mastering some simple BASIC programming, finds himself, or herself, asking questions about the machine:

- What is a string and how is it used?
- How can I create moving graphics?
- How can I interface external hardware?
- Why use machine-code?
- How can I link machine-code to BASIC programs?

The Timex Sinclair 1000/ZX81 User's Handbook answers these and many more questions covering binary and hexadecimal arithmetic, flowcharts, logic, graphics, the Z80A microprocessor, and machine-code. A glossary of over 100 terms is also included.

HOWARD W. SAMS & CO., INC.
4300 West 62nd Street, Indianapolis, Indiana 46268 USA

$5.95/22012

ISBN: 0-672-22012-1